AD-A049 483 ARMY ELECTRONICS COMMAND FORT MONMOUTH N J F/G 9/2
COMPUTER FAMILY ARCHITECTURE SELECTION COMMITTEE FINAL REPORT. --ETC(U)
SEP 77 M BARBACCI, R GORDON, R HOWBRIGG

UNCLASSIFIED ECOM-4529 NL

1 OF 3
AD
A049483

Research and Development Technical Report

ECOM - 4529

COMPUTER FAMILY ARCHITECTURE SELECTION COMMITTEE - FINAL REPORT, VOLUME IV - ARCHITECTURAL RESEARCH FACILITY; ISP DESCRIPTION, SIMULATIONS, DATA COLLECTION

Robert Gordon
Rosemary Howbrigg
Naval Underwater Systems Center

Susan Zuckerman
Naval Research Laboratory

Mario Barbacci
Daniel Siewiorek
Carnegie-Mellon University

September 1977

ECOM
US ARMY ELECTRONICS COMMAND FORT MONMOUTH, NEW JERSEY 07703

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The citation of trade names and names of manufacturers in this report is not to be construed as official Government indorsement or approval of commercial products or services referenced herein.

Disposition

Destroy this report when it is no longer needed. Do not return it to the originator.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: (14) ECOM-4529 (9) Research and development technical repts,
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): (6) Computer Family Architecture Selection Committee Final Report, Volume ~~IX~~. Architectural Research Facility; ISP Description, Simulations, Data Collection.
5. TYPE OF REPORT & PERIOD COVERED
6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s): (10) Mario Barbacci ~~(CMU)~~, Daniel Siewiorek ~~(CMU)~~, Robert Gordon, ~~(CMU)~~, Susan Zuckerman ~~(CMU)~~, Rosemary Flowbrigg ~~(CMU)~~
8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Center For Tactical Computer Sciences (CENTACS), DRSEL-NL-BC, Fort Monmouth, N.J. 07703
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: (16) 1L7_62701_AH92 ~~01~~09 (17) EJ

11. CONTROLLING OFFICE NAME AND ADDRESS
12. REPORT DATE: (11) ~~September 1977~~
13. NUMBER OF PAGES: 217 Pages (12) 225 P.

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): CENTACS, DRSEL-NL-BC, Fort Monmouth, N.J. 07703
15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report): Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES: This report consists of a summary and nine (9) volumes. It is the result of joint Army/Navy work. The complete report may be separately published by the Naval Research Laboratories.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number): Computer Family Architecture, ISP Instruction Set Processor ISPL, Instruction Set Processor Language, Register Transfer, ISPL Compiler, Architecture Research Facility.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This volume describes the process of automatically gathering architectural data from the benchmark programs. Formal descriptions of the candidate architectures were written and run under a simulator. Assembly listings of the benchmark programs were used to generate simulation command files containing absolute code. These command files were then used to initialize the simulated memory. The result of the simulated benchmarks were collected into a data base for post-processing. Automating the data collection process not only eliminated tedious and potentially error prone hand calculations, but also provided the means to

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

037 620 JOB

20. gather dynamic program behavior information that would be almost impossible to calculate manually.

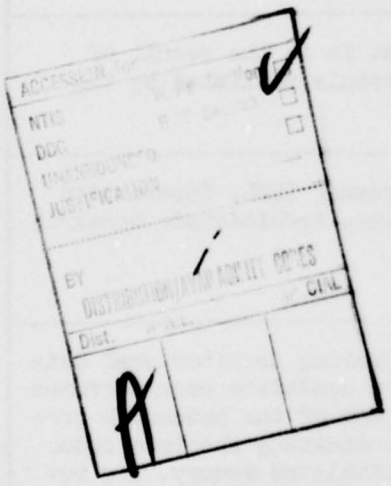


TABLE OF CONTENTS

SECTION

1. USES OF A FORMAL ISP DESCRIPTION

Digital systems can be viewed as a hierarchy of levels: electronic circuit, logic-combinational and sequential, register transfer, programming, PMS (Processor, Memory, Switch), and network. For each of these levels there is a need to be formal, for communication purposes, and to have a representation or language that is convenient to use in the design process, so that concepts can be stated easily and analysis can take place.

ISP (Instruction Set Processor) was first introduced by [Bell and Newell, 1971] as a language for the programming level. Its initial goal was to describe computers in a systematic way and provide the reader with the information required to program the machine, excluding implementation details (e.g., memory speed, data path organization, etc.). Thus ISP can be used to describe a machine's architecture, as it is defined by the Computer Family Architecture (CFA) Selection Committee.

There are several uses of a formal architecture description in ISP. A sampling of some of these uses that are relevant to CFA are listed below:

a. Simulator. The architecture can be simulated/emulated and used to write and debug benchmark programs. Moreover, depending upon the simulator capabilities, it can be used to develop support and application software, and as a training device.

b. Architectural Evaluation. The ISP can be "instrumented" so that the relative efficiency of a candidate can be measured as a function of architectural parameters (e.g., the S, M, and R measures used by the Committee and described in Volume III). These architectural parameters are a function of the dynamic behavior of the benchmark programs. Such dynamic behavior is tedious and error prone to calculate by hand. Moreover, in some cases the dynamic behavior of a program may be impossible to model analytically and hence must be measured by instruction traces. Thus an instrumented simulator was the easiest, most accurate method of measuring dynamic program behavior.

c. Experimentation. Once the ISP is written, only moderate effort is required to make a perturbation to the description. Thus, effects of architectural changes can be debugged, measured, and studied without committing any funds to hardware development.

d. Procurement. Since the ISP is a concise definition of an architecture, it can be used as the basis of a procurement document for its implementation.

e. Verification. The ISP simulation of an architecture can be used as a standard to verify the correctness of a hardware implementation of the architecture. This is done by running verification programs on both the ISP simulation and the hardware implementation and comparing results.

Based on its advantages in the evaluation phase and its continued usefullness throughout the CFA project, ISP was selected to describe the three final candidates. Section 2. outlines ISP while Section 3. details the ISP simulator. Writing of the candidate ISPs and the benchmark program data collections are treated in Sections 4. and 5., respectively. Section 6. gives some further details and reading hints on the candidate ISPs. Finally, Section 7. depicts the future role of ISP in CFA. Appendix A is an ISP Compiler and Simulator User's Manual while Appendix B contains the ISPs for the three final candidates. Appendix C contains a sample session in which the simulator is used to execute one of the benchmarks.

2. WHAT IS ISP?

The ISP notation was developed to formalize the information normally given in basic machine manuals and if possible to supplement and eventually replace what are known as "programming reference manuals." Hence the essential requirements were of readability, completeness, flexibility, and brevity.

a. Architecture vs. Machine Organization

In a hierarchy of computer system descriptions there exists a level, the programming level, in which the basic components are the machine instructions, operations, and the interpretation cycle, all of which are defined in terms of lower level primitives, the so called Register Transfer Level.

The separation between a description and its lower level realization permits the design of "computer families" i.e. multiple realizations (mappings) of the same high level description in which the behavior of a processor is determined by the nature and sequence of its operations. This sequence is given by a set of bits in primary memory (a program) and a set of interpretation rules (a central processor). Thus, if we specify the nature of the operations and the rules of interpretation, the actual behavior of the processor depends on the initial conditions and the particular program. During the execution of a program, some set of bits (an instruction) is read from the main memory into an instruction register located in the central processor. This set of bits then determines the immediately following sequence of operations. After this sequence has occurred, the next instruction to be executed is determined and obtained, and the entire cycle repeats itself. This interpretation cycle is performed by a part of the processor called the interpreter.

Computers are usually described in ISP in terms of the following relatively fixed format:

(1) Memory - Physical components which hold information encoded in data. Among others, we have: Primary-Memory to hold programs and data, Processor-State and General Registers, Console-State to interface the processor with the operator, Input-Output-State to interface the processor with external devices.

(2) Data-Types - Which are described in terms of registers which could carry information.

(3) Data-Operations - Defining data transformations that can be carried out in terms of data-types.

(4) Instruction-Format - Specific instances of Data-Types.

(5) Interpreter - The mechanism of the processor which fetches, decodes, and executes the instructions.

(6) Instruction-Set - The definition of the particular instructions that the processor executes.

This modularization of the description allows the designer to divide the processor in conceptually independent units - the actual hardware may or may not be implemented in that way.

b. Implementation Dependencies

ISP can be viewed as a programming language for certain class of algorithms i.e. Instruction Sets Processors (Architectures). Ideally, a language to describe architectures would not require the specification of any implementation details. Unfortunately, the ISP language does not aid the person writing the architecture description in distinguishing between truly architecture related items and implementation related items. The situation is similar to a common event in programming: A programmer describing an algorithm in a high level language is forced to take into account "implementation" details that are not part of the algorithm, e.g., the word length of the machine in which the program runs (it affects the result of the arithmetic operations). Due to the preciseness of ISP, it is necessary to define actions and registers which are not technically part of the architectural description. The mechanisms for doing functions, such as fetching instructions from memory, are not seen by the programmer. All she or he sees is the execution of an instruction or the trapping of error conditions such as exceeding memory boundaries. The methods for fetching the instruction, determining what it means, and starting the execution are left to the implementor. The only thing that the programmer is aware of is whether or not the instruction executes its given function and how it might fail. A complete and usable ISP description requires more than just the features a programmer would see.

3. SIMULATOR CHRONOLOGY AND CAPABILITIES

a. Chronology

During the spring of 1975 the Naval Research Laboratory contacted the Computer Science Department at Carnegie-Mellon University and expressed interest in the use of the ISP notation to write the formal specification of the CFA. CMU had been involved for several years in the development and use of ISP in design automation applications.

During the summer of 1975, NRL initiated the design of a system that would permit the "instrumented" simulation of the candidate architectures. The simulation facility was developed in several phases to minimize the risk of delays in the completion of the project. The compiler/simulation system was christened Architecture Research Facility (ARF) and is available on a nationwide basis through the computing facilities at NRL and CMU (the latter are available through the ARPA computer network, ARPAnet).

For the purposes of the selection process, the first three phases of ARF are of interest. ARF I was the enhancement of the ISP compiler available at CMU. This included expanding internal tables to allow the handling of large computer description, the implementation of better diagnostic facilities, and the addition of new features to the language. This phase was essentially completed by the end of 1975.

ARF II was a simulation facility based on an existing ISP simulator developed at CMU and was used to gather statistics from the benchmark programs. ARF II will be the facility described in the remainder of this section.

ARF III was a completely new simulation facility designed and implemented at NRL. It was completed and entered the testing phase while the selection committee was making its final decision. ARF III and its successors will continue to evolve and will constitute the basic tools needed for the modeling and verification of the chosen architecture in future CFA work.

b. Capabilities

A simulator is formed by linking the output of the ISP compiler with a table interpreting program. The simulator accepts commands from a teletype or user designated command file. The state of the simulator can be dumped to a command file which can be read at a future date when the simulation is continued. Command files can also be generated for initializing the target machine memory to contain a benchmark program (Section 5. describes how assembler output files were transformed into simulator command files).

The user interacts with the simulator through variable names and labels used in the ISP. The user can start and break the simulation on a label name. After a breakpoint the simulation is resumed on a continue command. Variables can have their values displayed or set under user control when the simulation is halted. During simulation, the successive assignments of values to traced variables are displayed on the user's terminal. Tracing allows the monitoring of the progress of a simulation and provides a very powerful debugging aid during the testing phase of an ISP.

For a complete description of the compiler and simulator capabilities the reader should consult Appendix A. A simple example of the use of the simulator is shown in Appendix C.

4. THE CANDIDATE ISPs

As part of the selection process, the Naval Research Laboratory funded a limited number of projects with the purpose of producing formal ISP descriptions of the candidate architectures. Carnegie-Mellon University has had many years of experience in computer description languages, particularly ISP, and provided expertise and tools for the task. The participation of CMU started during the spring of 1975 when initial plans were drawn for the implementation of a simulation facility that was to be used to verify the ISP descriptions and to execute selected benchmark programs. This simulated execution was to be fully instrumented and the measures obtained were used to compute the R and M measures of the candidate architectures.

a. . ISP Seminars

Following the December CFA meeting (Fort Monmouth) it was decided to organize a seminar with the purpose of familiarizing the CFA committee members with the use of the notation. The seminar took place at CMU during the 14-16 of January 1976. Eighteen people participated, representing many of the member organizations. A second, smaller, seminar took place at CMU during the 25-26 of May 1976. This meeting was oriented specifically to discuss the ISP description of the three final candidates.

In addition to the formal seminars, several informal meetings took place at CMU and NRL. During these meetings plans for writing the ISP descriptions were considered and problems and programming techniques were studied.

b. Division of Labor

During the preliminary phases of the selection process it was decided that the task of producing an ISP description for a candidate architecture was the responsibility of the organization or subcommittee proposing the architecture. Early efforts from some members of the committee allowed some of the ISP descriptions to be initiated before the final candidates were selected. The Naval Underwater Systems Center (New London) was sponsoring the candidacy of the IBM S/360-S/370 architecture and had already done some work on the description of a small 16-bit version of the IBM S/360. Dr. Robert Gordon and Ms. Rosemary Howbrigg of NUSC worked during the spring of 1976 on the full IBM S/360 description and by the time the final candidates were chosen the ISP description was close to completion. Dr. Daniel Siewiorek of CMU had completed a PDP-11 ISP description during the summer of 1975. This undertaking was part of CMU's work on computer description languages and applications. The PDP-11 description was subsequently modified and expanded according to the CFA requirements for statistic collections. The Interdata 8/32 presented a different picture. Ms. Susan Zuckerman of NRL started working on the ISP description of the 8/32 as late as April of 1976, but taking advantage of the accumulated experience on large ISP descriptions, particularly the IBM S/360-S/370, the Interdata 8/32 description was completed on time and the benchmarks for all three machines were processed before the 23 of July 1976 deadline.

c. Correctness of the ISP Descriptions

Perhaps one of the most serious questions to be asked is how correct are the ISP descriptions used in this project. The answer lies in the source of the information used to prepare the descriptions. All manufacturers provide

a "Principles of Operations" manual to aid the programmers. This manual is supposed to contain the true specification of the architecture. By far, the best documentation of the candidate architectures was provided by IBM, the least complete specification was that of the Interdata. Within IBM, the description of the architecture is the "Principles of Operation" manuals (i.e., an English description). DEC provides several manuals, one for each model, and this required going back and forth between manuals when details were not clear. The description of the Interdata required consultation with the manufacturer and some of the information was not guaranteed to be valid for later versions of the 8/32. We used the same documents, but being humans it is possible that we interpreted the English definition of the architecture differently from the implementors. All test cases which were run did agree with identical test cases run on real machines at Carnegie-Mellon University and Interdata. It must be remembered, though, that an ISP description is just another computer program and thus, if it is to be used, must be verified as being correct. This will require additional documentation of proprietary nature and architectural verification programs that the manufacturers have. This will have to be handled during the implementation phase, for the selected architecture.

5. DATA COLLECTION

Writing an ISP description for a large machine is not a trivial task. The candidate architectures had large instruction sets and although some features were excluded from the ISP, writing many hundreds of lines of code in a short period of time was a very satisfying and remarkable achievement when compared with software projects of similar magnitude.

In order to complete the task on time certain features of the candidate architectures were not described. These features were omitted on the basis of their importance to the CFA data collection phase. However, the ISP of the selected CFA will be fully specified and will contain all such features.

a. Memory Management - All three architectures have a virtual memory management mechanism, described in their principles of operations manual. By common agreement among the three architecture subcommittees this was considered a subsetable feature. The PDP-11 description already had this feature and was later used in one of the benchmarks. Neither the IBM S/360 nor the Interdata 8/32 descriptions have it.

b. Decimal Instructions - Only the S/360-S/370 offers this option. It was not needed to run the benchmarks. By common agreement among the architecture subcommittees it was deemed subsetable and therefore not included in the S/360-S/370 description.

c. Floating Point Instructions - All three architectures offer a Floating Point Instruction Set. Including this feature in the description would have greatly increased the time and manpower requirements for the task. The FP instructions are among the most complex instructions of any machine. By common agreement between the architecture subcommittees, floating point instructions were not included in the ISP descriptions. However, since some of the benchmarks required floating point operations, dummy procedures were included in the descriptions. This served a dual purpose, first it allowed us to keep the correct counts needed to compute the R and M measures, and second, it allowed the detection of those places where a benchmark executed a floating point operation and had to be helped around the trouble spot via simulation commands.

d. Error Handling - All three architectures define certain error recovery procedures (e.g., handling illegal operation codes, detecting address boundary errors, etc.). This feature was considered not crucial (the benchmarks were for the most part working programs that had already been executed on real machines) and it was up to the ISP writers to include it or not. The S/360-S/370 description contains a complete error handling mechanism, as defined in the principles of operations. The Interdata 8/32 description also has some error detection and recovery mechanisms. None were included in the PDP-11 description.

a. Final Debugging

All three descriptions were developed on the time-sharing facilities at CMU. The Advanced Research Projects Agency Computer Network (ARPANET) provided long distance access and the descriptions of the S/360-S/370 and the 8/32 architectures were written, debugged and tested directly from NRL and

NUSC (New London). For the final testing and running of the benchmarks, the people responsible for the descriptions met at CMU and the collection of statistics was performed in Pittsburgh.

Although most of the benchmarks were debugged and run on the real machines, other benchmarks were executed exclusively under the simulator. The latter included those programs using privileged instructions that were not directly available to non-system programmers (e.g., interrupt and I/O handlers). For the former set of benchmarks, results from the actual runs were available and used to check the simulated execution. For the second class of benchmarks the tracing and single stepping facilities of the ISP simulator were used to verify the correct execution of the programs. Breakpoints were used to detect the execution of non-implemented instructions (e.g., the Floating Point Set) and the simulated execution was guided around these instructions, taking care that the machine status and condition codes were properly set.

Although the ISP descriptions were essentially debugged before the benchmark execution phase started, there were some minor modifications and corrections that had to be done. These were performed concurrently with the data collection phase. The largest unforeseen problem was presented by the memory management feature of the PDP-11 which was based on the PDP-11/40 and had not been tested. One of the benchmarks (Quick Sort) called for a large address space and required the enabling of the feature. Unfortunately, the benchmarks had been tested on a PDP-11/45 which uses different unibus addresses for the memory management registers and this required minor modifications to the benchmarks. Most other problems were of a simpler nature and required only a few minutes to correct. It should be noted here that the simulator facility was also used to debug some benchmarks for the Interdata 8/32 before they were executed on the real machine. This was because no 8/32 was available near CMU and a large turn-around time (several days) would have complicated the debugging of the benchmarks.

b. Preparation of Simulation Benchmarks

The ISP simulator provides commands for the loading and initialization of the simulated machine memory and internal registers. The single most important feature of the command language which permitted the fast execution and collection of statistics was the ability to read "command" files containing the benchmarks to be executed. The command language can not handle programs in symbolic form (assembly language) and requires the preassembly of the programs into absolute, numeric, code. To this effect, a set of programs was developed at CMU which permitted the translation of assembly listing prepared by the real machine assembler into simulation command files. The operation was performed off-line.

Three sets of programs were prepared, one for each candidate architecture. The assembly listings were transported to CMU's PDP-10 using magnetic tapes (for the S/360 and the 8/32) or were prepared directly on the PDP-10 using a cross-assembler (for the PDP-11). The format of the assembly listings is different for all three machines. Nevertheless, in all three cases, it contains a listing of the relocatable object code. The procedure to translate this relocatable code into simulation command files consisted of the isolation of the code, the modification of the relocatable addresses using a user specified base address (multiple base addresses can be specified for the different control sections of the S/360), and the generation

of the ISP simulator commands loading the simulated machine memory locations with the code.

A total of 114 simulation runs were executed. They correspond to a total of 70 different benchmarks (some benchmarks called for several test cases, in other instances a benchmark had to be divided into separated sub-cases). The 70 benchmarks were divided as follows: 26 for the PDP-11, 22 for each of the IBM S/360-S/370 and Interdata 8/32. The appendix includes several examples of the command files used to simulate the benchmarks.

c. . Counter Setting, Dumping, and Data Reduction

The ISP simulator permits the instrumentation of an ISP description by associating activity counters with each of the machine registers and memories. These counters allow the collection of statistics indicating the number of times each component of the machine is read from or written into. A normalized count is used and the counters are updated in terms of the number of 8-bit bytes actually involved in the operation. For registers with length different from a multiple of 8 bits the length count is rounded up (i.e., a 10 bit register operation counts as 2 bytes). A separate counter is kept for each label in the ISP description. Labels are included in the ISP descriptions to identify machine instructions, addressing modes, loops (used to describe vector like instructions such as move character (MVC) on the S/360), as well as other ISP procedures. During the execution of the benchmarks, a data base was created by collecting dumps of the counters after each benchmark was completed. The files containing the counters were then processed by other, off-line, programs in order to arrive at the M and R measures.

d. Artificial Labels in the ISP Descriptions

Certain modifications not normally needed were made to the ISP descriptions to aid in the collection of data during the running of the benchmark programs for the CFA project. Several labels and "do-nothing" procedures were added to allow easier measuring. These should not be looked at as necessary for the architecture description. A typical example of the need for the extra labels is given in the RX instructions of the S360: Register [0] can not be used as an index register and it was necessary to count the number of times that Register [0] was being specified as the Base or Index register. The labels added to count these events are clearly not part of the architecture or even the organization. Certain items, such as modifying the program counter during a branch operation or the setting of condition codes as a result of an instruction, were not to be measured in any of the three architectures for the CFA project. This required the addition of artificial labels that were used to identify portions of the description during which counting of events was disabled. This was typical of those actions which in a reasonable implementation would be done using ad-hoc circuitry, aside from the main operational units of the machine and thus, were not considered to affect the R measure.

6. THE CANDIDATE ISPs - READING HINTS

a. The ISP Description of the IBM S/360

In writing the ISP description of the IBM S/360 family of computers, a subset of the architecture was chosen. It included the entire standard instruction set, the protection feature instructions, and the direct-control feature instructions. It excluded floating-point and decimal instruction definitions. These choices were made due to limitations of time and personnel. It should be noted that IBM markets four instruction sets in the S/360 line. These are the Standard set, the Commercial set (Standard plus decimal), the Scientific set (Standard plus floating-point), and the Universal set (Standard plus decimal plus floating-point plus storage protection). Timer and direct-control features are additional options. Due to the upward compatibility between the IBM System/360 and the IBM System/370 lines, the generated description could be expanded to achieve the IBM System/370 description, but the level of effort required for this would be substantial.

For the purpose of extracting data for CFA decisions and for increasing the ease of running the simulator, several additional choices were made.

(1) The diagnose instruction, which has a model dependent definition, was not written to directly correspond to any particular model. It was modified and used in aiding the termination of a simulation run.

(2) Since several benchmark program authors wanted to use the compare logical long (CLCL) instruction of the IBM System/370 architecture, it was added to allow for collecting data, but was not a true description of the instruction since it was not written as an interruptible instruction.

(3) The test and set instruction was not described since no adequate mechanism in the ISP simulator allowed for a true execution of the mechanics of the instruction within one ISP program. A second parallel process should be defined for the main memory control unit. The same holds true for the I/O channel definition.

(4) The front panel was minimal. Only a stop/run switch was included. Initial Program Loading (IPL), which is a front panel function, was not described.

Information necessary to write the description was obtained from the "IBM System/360 Principles of Operation" and the "IBM System/370 Principles of Operation" manuals. For the subset of the architecture described, it was not necessary to request further assistance and explanations from the manufacturer. Side effects of instructions were adequately described in the manuals. Model dependencies were also clearly enumerated. Instruction formats and addressing mechanisms were well defined and logically constructed. No ambiguities were discovered that couldn't be resolved using only the "Principles of Operation" manuals. This is not intended to imply that it would not be necessary to get further clarifications from IBM when describing the more complex supervisor state instructions in the System/370. In addition, we noted that some of the privileged state features of the System/370 are very model dependent and will probably be more so in the future. IBM may maintain compatibility at the problem state level only.

For reading ease, this ISP description consists of several sections:

Section 1: The first section contains declarations and is divided into two areas. First are the declarations which are part of the architecture description (i.e., those seen by a programmer). Next are implementation related variables which were items needed to adequately describe the architecture in ISP but are not seen by a programmer. It was divided in this way since the ISP language makes no distinction between truly architecture-related items and items necessary for a complete simulation of the architecture.

Section 2: The second section contains utility routines which were used throughout the description. Some routines are implementation related if they use implementation related variables.

Section 3: The third and largest section contains operand address generation routines and instruction descriptions. The instruction set is divided into four groups, each having a different amount of address generation required.

Section 4: The fourth section contains the interrupt processing description of the architecture. The order of handling the different classes of interrupts and the actual processing is thoroughly described in the IBM manuals. This is very unique in an architecture description from a manufacturer. A user gains a sense of reliability about the system, knowing that the real sequence of events that will occur on an interrupting condition allows the machine to recover from certain simultaneous hardware and software faults.

Section 5: The fifth section contains the instruction decoding and instruction cycle routines which fetch and execute an instruction. Everything to this point is considered to be declarations in the ISP language. Since all procedures must be defined before they are referenced, the last and smallest section contains only the executable program. The execution consists of doing one instruction and checking for interrupts whenever the CPU is not stopped; then repeating the cycle.

b. The ISP Description of the Interdata 8/32

The Interdata 8/32 ISP description is organized into seventeen sections. The description omits the I/O instructions, the floating point and double precision registers and instructions, and the MAC (memory access controller) operations. The ISP description is considered as both a simulation program (and therefore structured) and as a machine specification description.

Section 1: Interdata 8/32 storage resources as seen by the systems programmer (memory organization, register sets, program status word, instruction register).

Section 2: Temporary registers used for ISP description and implementation.

Section 3: ISP common subroutines used in later instruction descriptions.

Section 4: Interdata instruction format routines.

Section 5: Illegal instruction handler.

Section 6: Interdata LOAD and STORE instruction descriptions.

Section 7: BOOLEAN instructions.

Section 8: SHIFTs, TEST & SET, and TRANSLATE instructions.

Section 9: COMPARE, and CONVERT to HALFWORD VALUE instructions.

Section 10: BIT manipulation instructions.

Section 11: Arithmetic instructions.

Section 12: BRANCH instructions.

Section 13: CIRCULAR LIST instructions.

Section 14: Privileged Instructions and SUPERVISOR CALL instructions.

Section 15: Unimplemented instructions (floating point, I/O, double precision).

Section 16: Emulation routines for ISP (IFETCH, IXQT, INTCHK)

Section 17: Main instruction loop: EMULATE.

The Interdata 8/32 Manual omits discussion of instruction effects when non-standard (unexpected) parameters are specified in an instruction. For example: What happens if an odd register is given and the instruction expects an even register? What happens if memory boundary addressing is not adhered to? Conversations with Interdata personnel were needed to clarify these and other questions. The ISP description reflects the Interdata 8/32 operations as specified by the manual and personnel.

c. The ISP Description of the PDP-11

The PDP-11 line consists of 13 different models. In general the models are upward program compatible but there are instructions implemented in low end models that are not implemented in high end models and vice versa. The Initial PDP-11 description was modelled after the 11/40, a mid-range machine. Subsequently, the PDP-11/70 was specified by the CFA selection committee as the official PDP-11 architecture to be evaluated. Therfore the ISP was updated to incorporate the extra instructions.

Several features of the architecture were omitted from the description, and this situation will have to be corrected in the future: The floating point instructions, the interrupt mechanism, and the error detection and recovery mechanism. It should be noted here that there exist two different floating point instruction sets in the PDP-11 line, and that the memory management facility is not homogeneous across different models. These discrepancies will be resolved and an 11/70-compatible architecture will be specified in the final, formal ISP.

Following is a page by page description of the ISP:

Page 2-1. The primary memory and mappings (note word/byte memory and I/O page), central processor registers, and the floating point processor status register.

Page 3-1. The PDP-11/40 memory management registers and error registers that allow an instruction retry.

Page 4-1. Temporary registers not seen by programmer. These registers are necessary to completely define the algorithms performed by the hardware (such as address calculation) but these registers are not part of the architecture.

Page 5-1. Instruction decoding formats.

Page 6-1. Start of the procedures. Memory accessing procedures.

Page 7-1. Effective address calculating procedures.

Page 8-1. Condition code setting procedures.

Pages 9 through 15. These are the actual instruction definitions. Similar instructions are grouped together into classes that follow the several levels of decoding that the hardware must go through.

Page 16-1. The instruction interpretation cycle.

During the course of the benchmark debugging and data gathering, several benchmarks made use of instructions not previously described in the ISP. These added instructions included SOB, MUL, DIV, ASH, and ASHC.

7. FUTURE USES OF ARF IN CFA

As stated in Section 1., Uses of a Formal ISP Description, ARF will continue to play a role in architectural experimentation, development of a procurement document, and verification of implementations of the CFA architecture. In addition, ARF and other ISP driven software tools are finding an expanding use in the DoD and ARPA community. Some of these uses include:

a. An emulator facility developed for the Air Force by the University of Illinois compiles code for a PDP-10 directly from an ISP description. An Intel 8080 runs at about a 300:1 simulation ratio. The facility will be used to debug and monitor large tactical programs for existing machines. It will also be possible to write code for machines before the hardware is available for users.

b. Even faster emulation speeds are possible when using microcode. TRW compiles microcode for a QM-1 from a SMITE description. They have achieved an 11:1 simulation ratio for an Intel 8080. SMITE is an ISP-like language and a program is being written to translate ISP descriptions into SMITE.

c. The potential for tools that operate relative to a computer description could represent a significant breakthrough in the manner that computer systems (hardware/software) are designed and evaluated. Currently effort is underway at Carnegie-Mellon University to develop a hardware design automation program and a compiler-compiler that take as input the symbolic description of a computer. Early results indicate that the resultant hardware design and generated code will be comparable to those produced by hand. Effort is also underway at Yale to automatically generate assemblers and I/O device handlers from computer descriptions.

d. Other areas in the early stages of development include automatic diagnostics generation, microcode generation, machine verification, and high level performance/reliability evaluators.

In the next five to ten years one can envision a system of programs that take as input computer descriptions and language and problem specifications, and from these, generate operating systems, compilers, and other support and application software automatically. Thus the entire proposed architecture could be evaluated without committing too many years of effort in both hardware and software design.

It is hoped that, with the software tools already developed, formal computer descriptions will play an increasing role in the Department of Defense's evaluation, procurement, verification, and programming of computers.

APPENDIX A

The Symbolic Manipulation of Computer Descriptions: ISPL Compiler and Simulator

Mario R. Barbacci
Department of Computer Science
Carnegie-Mellon University
Pittsburgh Pa.
August 2, 1976

This project is supported in part by the Advanced Research Projects Agency (ARPA) of the Department of Defense, under contract F44620-73-C-0074, monitored by the Air Force Office of Scientific Research, and by the National Science Foundation, under grant GJ 32758X.

TABLE OF CONTENTS

A User's Guide to the ISPL Compiler

Mario R. Barbacci

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa.

ABSTRACT

The compiler described in this manual will translate programs written in a subset of ISP [Bell, 1971] into register transfer level instructions. The code thus generated could be used for the implementation of wiring list generators, simulators, or other Computer Aided Design applications. This manual describes the syntax and semantics of the language (ISPL) accepted by the compiler.

ACKNOWLEDGEMENTS

The compiler described here is an improved version of an original system implemented by S. Goldman and R. Scroggs. The syntax graph driving the compiler is generated using a program (GRPGEN) written by P. Karlton and R. Scroggs. This version of the manual reflects the modifications and improvements suggested by the users during the preparation of the ISP description of the candidate architectures for the Army/Navy CFA project. Special thanks are due to H. Elovitz (NRL), R. Gordon (NUSC), R. Howbrigg (NUSC), D. Siewiorek (CMU), and S. Zuckerman (NRL).

The Symbolic Manipulation of Computer Descriptions:
ISPL Compiler and Simulator

The Department of Computer Science at Carnegie-Mellon University is currently engaged in a research project exploring the uses of computer description languages in the automatic design of both software and hardware systems. This document describes a language, ISPL, based on the Instruction Set Processor notation of Bell&Newell [Bell,1971]. The language was designed as a tool for the description of instruction sets i.e. the architecture of a computer, and has been used extensively in a design automation project at CMU [Siewiorek,1976] and in the Army/Navy Computer Family Architecture Project.

Traditional computer description languages have been designed primarily for human communication and/or simulation. The SMCD [Barbacci,1974] project has the more ambitious goal of developing design automation tools which would permit the generation of *machine-relative* software, documentation, hardware modular design, program verification, simulation, and generation of microcode. As in any evolutionary project, preliminary results are necessarily short of the ultimate goal; thus at this point we can present two concrete systems: a compiler and a simulator. A machine-relative compiler-compiler is being investigated by a group under W. Wulf. An automatic generator of hardware modular specifications is being developed by a group under D. Siewiorek and A. Parker. Further studies of computer descriptive languages are being carried out by this author and others.

As indicated above, the systems described in this report have been used as part of the Army/Navy CFA project, sponsored by the Army Electronics Command and the Naval Research Laboratory. Part of the project involved the description, in ISPL, of three commercial architectures: The DEC PDP-11, the IBM /360,370, and the Interdata

8/32. These descriptions were used to collect statistics on the execution of a set of benchmark programs under the ISPL simulator. Although the simulator is not particularly fast, its interactive facilities allow very extrict control and detailed analysis of the register transfer operations being performed during the fetch/decode/execute cycle of the machines. The simulator was not meant to be used as a software development tool (although in fact, some CFA benchmarks for the Interdata 8/32 were debugged under the simulator, it being more accessible at CMU than the real machine), it is rather an Architectural Design tool that allows the user to explore alternative instruction sets and to collect statistics on the performance of the architectures.

Mario R. Barbacci
August 2, 1976

[Barbacci,1974] Barbacci, M.R. and D.P. Siewiorek: "Some Aspects of the Symbolic Manipulation of Computer Descriptions", Department of Computer Science, Carnegie-Mellon University, July, 1974.

[Bell,1971] Bell, C.G. and A. Newell: "Computer Structures: Readings and Examples", McGraw-Hill Book Company, New York, 1971.

[Siewiorek,1976] Siewiorek, D.P., and M.R. Barbacci: "The CMU RT-CAD System: An Innovative Approach to Computer Aided Design", National Computer Conference, NCC76, New York, June 1976.

1. Introduction

The ISP (for Instruction Set Processor) notation was developed for a text [Bell, 1971] to precisely describe the programming level of a computer in terms of its memory, instruction format, data types, data operations, and a set of interpretation rules.

The behavior of a processor is determined by the nature and sequence of its operations. This sequence is given by a set of bits in primary memory (a program) and a set of interpretation rules (usually in the central processor). Thus if we specify the nature of the operations and the rules of interpretation, the actual behavior of the processor depends on the initial conditions and a particular program.

Although the above format is commonly used to describe a digital computer, ISPL is not intended to force the user into a given description style; ISPL can be used to describe register transfer systems in general (digital computers are a subset of such systems, namely those systems that interpret an instruction set).

The subset of ISP implemented by the compiler under discussion contains a number of features that allow the user to describe a wide variety of digital systems: Pseudo register declarations, macros, and compound statements. For efficiency reasons, certain other features described in [Bell, 1971] are not implemented. Among these are: multidimensional memory arrays, parameterized procedures, multiple word access, and scattered bit access. However byte access is implemented.

An ISPL program consists of a description of the memory components (memories and registers) and a description of the behavior of the system. Memory components are defined in ISPL by a name and a description of their structure using brackets to

group the subcomponents along a given dimension. In the current implemention the only subcomponents allowed are memory words and bits (as subcomponents of memory words and registers). The behavior of the system is given by a set of register transfer statements. These statements can be performed in sequence or concurrently. In ISPL, concurrency of actions is the rule rather than the exception, and it is reflected in the use of ";" as a delimiter for lists of concurrent actions. Sequencing is expressed by using the term "next" as a delimiter for lists of sequential actions. Complex concurrent and sequential activities can be described in terms of simpler activities using "next", ";", "(", and ")" in an Algol-like block structure.

The ISPL compiler produces code for an idealized Register Transfer Machine. There are two types of instructions in the RTM: Data and Control instructions. Control instructions are used to sequence the operation of the machine. They contain instructions to START, STOP, BRANCH, DIVERGE into concurrent execution paths, etc. The Data instructions are used to define the Arithmetic and Logical operations among the registers of the machine. They are described in terms of a 3-address format:

destination ← source1 operation source2

The RTM code produced by the compiler is presented in two formats. The first format is simply a tabular listing intended primarily for human use. The second format is intended primarily for machine consumption. The human intended tabular representation could be digested by suitable string manipulating programs and stored into a more convenient machine format. Several reasons argued against this approach: depending on the language used, writting these interface programs might involve a non trivial amount of work. Worse yet, any format modification intended to help human readers will render these programs obsolete. The solution adopted was to produce

another copy of the RTM code directly into a machine understandable format. Thus the version of the RTM code intended for machine use is created as a "program" using MACRO-10 as the intermediate language. The format of these programs is described in the appendices.

2. Declarations

There are two types of declarations in ISPL: Memory Declarations (explained in this section) are used to describe the structure of the registers and memories in a machine; Procedure Declarations (explained in later sections) are used to describe the behavior of the functional units in a machine.

2.1. Memories and Registers

Memory components are defined in ISPL by a name and a description of their structure. The number of subcomponents at each level of decomposition is given by a bracketed list of constants, much like an array declaration in Algol.

declaration-part ::=	DECLARE ***declaration-list*** ERALCED
declaration-list ::=	***declaration*** |
	declaration-list ; ***declaration***
declaration ::=	***memory-declaration*** |
	memory-declaration := ***memory-declaration*** |
	procedure-declaration
procedure-declaration ::=	***identifier*** := (***statement-list***)
memory-declaration ::=	***identifier structure-declaration***
structure-declaration ::=	[***word-list***] < ***bit-list*** > |
	[***word-list***] < > |
	< ***bit-list*** > |
	< >
word-list ::=	***name-list***
bit-list ::=	***name-list***
name-list ::=	***element-range*** |
	name-list , ***element-range***
element-range ::=	***number*** | ***number*** : ***number***

The declarations are given by a list of individual component declaration using ";" as delimiter. There are two types of memory declarations: 1) A definition of a physical component (physical declaration), and 2) A definition of a logical component (logical declaration) in terms of a previously declared (physical or logical) component. A logical declaration uses the ":=" operator to make an equivalence between two components.

Examples

A<15:0>	Declares A as the name of a register 16 bits wide, named 15, ... 0 (from left to right). The ":" or range operator is used to denote an abreviated list of subcomponent names.
Mp[0:4095]<0:11>	Square brackets are used to specify those dimensions where the accessing is done through some "addressing" (switching) schema. The memory, Mp, consists of 4096 words, each of 12 bits, named (from left to right) 0,1,...11.
R<15,13,11,9:10>	In general, the list of subcomponents along any dimension is given by a list of "names" for the individual subcomponents. Numbers used to name individual elements do not indicate relative position.

Mw[32767:0]<15:0>;

Mb[65535:0]<7:0>:=Mw[32767:0]<15:0>; Now the designer can use either Mw (the "word" memory) or Mb (the "byte" memory).

The only concession to the use of numbers as both names and position indicators is by using the range (":") operator, whereby the abreviated list consists of the bounds and all integers in between, with the implication that these consecutive numbers also name consecutive (from left to right) elements. The use of an empty bit-list (<>) indicates a single, unnamed bit.

Undeclared variables or multiple declarations of a variable are, usually, non-fatal errors. The compiler will warn the user it this situation arises. The compiler compares the lengths (Nwords*Nbits) of the left and right hand sides of a logical declaration; if the lengths do not match a warning is issued.

2.2. Macros

A different type of declaration, the MACRO declaration, allows the designer to

abreviate the description by naming often used strings of characters. The macro name can then be used instead of the full string. The format of a macro declaration is the following:

MACRO *identifier* := any-string-of-characters-not-containing-a-$-sign $

Macros are handled in its entirety by the lexical phase, thus the parser never "sees" a macro expansion. Macros can, therefore, be declared at any point in the description, not necessarily in the declaration part, and remain in effect until the end of the description.

Examples

MACRO SIGNBIT := ACC<0> $ — The use of SIGNBIT some time later in the description is equivalent to using ACC<0>. Macros are strictly in-line string substitutions.

A macro can be defined in terms of other macros and the user should be careful to avoid a recursive definition which would create a non-terminating string replacement loop.

There are implementation dependent limits on the size of a macro string. If a macro declaration exceeds this limit (1000 characters at present) a warning will be issued. Results might be unpredictible if this situation occurs.

2.3. Identifiers and Constants

An identifier in ISPL is a string of letter, digits, and "."'s, beginning with a letter; the "." is included as an identifier character for readability purposes. In the current implementation only the first 6 characters of an identifier are kept by the compiler. Identifiers must, therefore, differ in the first 6 characters for the compiler to distinguish them. The lexical phase accepts upper and lower case ASCII characters but

they are converted and stored internally as upper case characters. This is another limitation of the implementation.

For readability purposes, identifiers can be followed by a larger and more descriptive version of the identifier. This secondary identifier is treated like an inline comment by the lexical phase. The syntax for this extended identifier use is:

short.identifier\this.is.a.long.identifier

An extended identifier can be appended to a short identifier using the "\" character. Such compound identifiers are valid wherever an identifier is valid. Notice that this is not the same thing as an "alias", as described in the full language [Bell, 1971]. The secondary name is stripped by the lexical phase and the designer must use the primary name for identification purposes.

Constants are strings of digits, interpreted as a number in some base. The default base is 10 (i.e., constants are decimal numbers unless otherwise specified). Constants in base 8 (octal numbers) must be tagged with the character *, as in *100 (decimal 64). Constants in base 2 (binary numbers) must be tagged with the character ', as in '100 (decimal 4). Constants in base 16 (hexadecimal numbers) must be tagged with the character ", as in "A1 (decimal 161). The length of a constant is the minimum number of bits needed to represent it (i.e. leading 0's are stripped). The constant 0 is 1 bit long. The current implementation of the compiler limits constants to a maximum size of 35 bits.

2.4. Comments

Comments can be inserted in a description by preceeding the comment string with the character "!". All characters following the "!" until the end of the line are ignored.

3. Register Transfers

Register Transfers are used to describe the data operations on the memories and registers (the data components) of the system. The syntax of a transfer follows very closely that of most programming languages. The main difference is the use of some special operators and the use of a non-standard operator precedence to accomodate these new operators.

The operators act upon the components of the system by taking the data stored in some components (the inputs), operating (i.e., transforming) on the data, and storing the resulting data in some component (the output).

The data used by the operators is defined in terms of the components that contain it. Since the memories and registers are declared as structured components made out of words and bits, a structure selector is needed in order to access or store data.

3.1. Structure Selectors

structure-selector ::=	***term*** | ***term*** < ***selector-range*** >
term ::=	***number*** | ***memory-access*** | (***expression***)
memory-access ::=	***identifier*** |
	identifier [***arithmetic-expression***]
	identifier [***element-name***]
element-name ::=	***number***
selector-range ::=	***bit*** | ***bit*** : ***bit***
bit ::=	***number***

The ***terms*** are the building blocks used in a register transfer expression. A ***term*** can be a constant, a ***memory-access*** (to select data stored in a memory or register), or an ***expression*** in parenthesis (thus allowing large and complex register transfer expressions).

A ***structure-selector*** is used to select parts of a ***term*** (i.e. to select bits of a register, a constant, or an expression). The nature of the register transfer operators requires that the operands be of homogeneous type (i.e., register-like) and length. Thus multiword memories must be accessed using an ***arithmetic-expression*** (the address calculation) enclosed in "[" and "]" to select one and only one word of the array.

The compiler compares the maximum value that the result of an address computation can have with the number of words declared for a memory. If the former exceeds the latter, a warning is issued.

When a ***selector-range*** is applied to a memory or register access term it must use the bit names used in the declaration. When it is applied to other types of term, whose structure has not been declared (i.e., constants and expressions), the bits of the term are implicitly named n, n-1, ..., 1, 0 (from left to right).

Examples

ACC	Select the entire ACC register
Mp[Pc]	Select the word whose address is contained in register Pc
ACC<5>	Select bit 5 of register ACC
Mp[R[INDEX]+DISPLACEMENT]<0>	Select bit 0 of the word whose address is given by the effective address calculation expression
(A<7:0>+B<7:0>)<5:4>	Select the 5th and 6th bits (from the right) of the result of the addition

Attempting to access undeclared bits of a register or memory word will result in a warning message. The compiler will then default the erroneous bit name to the leftmost bit of the declaration. When the selector range of a register or memory word attempts to switch the relative position of two bits, the compiler will switch the

selector range boundaries and issue a warning message. For instance, if X is declared as X<0:5>, both X<2:3> and X<3:2> are equivalent terms but in the second case a warning is issued.

3.2. Transfers

Register transfers are used to modify the contents of the registers and memories. The syntax of a transfer is the following:

transfer ::= ***memory-access*** ← ***arithmetic-expression*** |
memory-access <***selector-range***> ← ***arithmetic-expression***

The use of a ***selector-range*** on the left hand side of the "←" specifies a partial register (or memory word) modification; the non-selected bits are not disturbed. If the right hand side is shorter than the left hand side, the result is stored right justified and 0's are concatenated to its left to clear the high order bits of the left hand side. If the right hand side is larger than the left hand side truncation of the high order bits will occur (the compiler will issue a warning if this situation occurs).

The right hand side of a transfer is always an ***arithmetic-expression***. The difference between an ***arithmetic-expression*** and an ***expression*** properly is in the use of relational operators, which are not allowed in the former. We will give more details in the subsection dealing with expressions.

3.3. Shift Operators

shift ::= ***structure-selector*** |
structure-selector shift-op structure-selector

shift-op := ↑SL | ↑SR | ↑SL0 | ↑SR0 | ↑SL1 | ↑SR1 | ↑RL | ↑RR |
concatenation

concatenation ::= @

A ***shift*** is the first step in the hierarchy of register transfer operations, shift operators have the highest binding power (precedence). A ***shift*** always takes the following form:

left.operand ***shift-op*** right.operand

The meaning of the operators (all of them have the same precedence) is the following:

OPERATOR	MEANING
↑SL	Shift left the left.operand, one position, and insert the (rightmost bit of the) right.operand into the vacant position, dropping the leftmost bit of the left.operand. The length of the result is the same as the length of the left.operand. The result can be stored in a register or used as an operand when building complex expressions. The operator does not modify the left.operand, only the transfer operator ("←") can perform side effects.
↑SR	Shift right the left.operand, one position, and insert the (rightmost bit of the) right.operand into the vacant position, dropping the rightmost bit of the left.operand. The length of the result is the same as the length of the left.operand.
↑SL0	Shift left the left.operand the number of positions indicated by the value of the right.operand inserting 0's in the vacant positions and dropping the righmost bits of the left.operand. The right.operand is treated as an unsigned integer. The result has the same length as the left.operand..
↑SR0	Similar to ↑SL0 but shifting right.
↑SL1	Similar to ↑SL0 but inserting 1's into the vacant positions.
↑SR1	Similar to ↑SL1 but shifting right.
↑RR	Rotate towards the right the left.operand by the number of positions indicated by the value of the right.operand. The length of the result is the same as the length of the left.operand.
↑RL	Similar to ↑RR but rotating left.
@	Concatenate the left.operand with the right.operand. This operator is included among the shift operators for symmetry reasons. The length of the result is the sum of the lengths of the operands.

3.4. Arithmetic Expressions

complement ::=	***shift*** | NOT ***shift***
conjunction ::=	***complement*** |
	conjunction AND ***complement*** |
	conjunction EQV ***complement***
disjunction ::=	***conjunction*** |
	disjunction OR ***conjunction*** |
	disjunction XOR ***conjunction***
negation ::=	***disjunction*** |
	- ***disjunction*** |
	MINUS ***disjunction*** |
	+ ***disjunction***
factor ::=	***negation*** |
	factor * ***negation*** |
	factor / ***negation***
sum ::=	***factor*** |
	sum - ***factor*** |
	sum MINUS ***factor*** |
	sum + ***factor***
arithmetic-expression ::=	***sum***

All logical operators (NOT, AND, EQV, OR, and XOR) operate on a bit by bit basis. If the operands have unequal lengths the shortest operand is expanded (on the left) with 0's.

The arithmetic operators, with the exception of MINUS, operate on unsigned (pure magnitude) operands, the MINUS operator assumes a Two's Complement represention with a sign bit in the leftmost position. The main difference is in the padding used to match the length of their operands. The MINUS operator extends the sign of the shortest operand, the other operators use 0 as the padding character.

The length of the result of the infix operators "+", "-", and "MINUS" is one bit larger that the largest operand. The length of the result of the "*" operator is the sum of the lengths of the operands. The length of the result of the "/" operator is the same as the length of the left operand (the dividend).

3.5. Relational Expressions

In order to describe non-trivial systems, ISPL provides certain facilities to control the execution of the transfers. Thus certain transfers may or may not be executed depending on the result of some previous operation. These conditional activities are described in more detail in the following section. Here we are concerned with the basic data operators of the language, among which we include the relational operators used to build conditional expressions.

relation ::=	***arithmetic-expression*** | ***arithmetic-expression relop arithmetic-expression***
relop ::=	EQL | NEQ | LSS | LEQ | GEQ | GTR | TST
expression ::=	***relation***

Relational operators perform a test between their left and right operands. The result for all these operators, with the exception of TST, is a boolean value (TRUE or FALSE) which can be tested by one of the control operations defined in the following section. All relational operators treat the operands as unsigned integers. A 2's complement representation of a negative number will therefore look greater than a positive number of the same length.

The TST operator performs a logical subtraction of its operands and produces a result of 0, 1, or 2, indicating that the left operand is less than, equal to, or greater than the rigth operand, respectively.

Beware that relational operators have less precedence than logical and arithmetic operators, thus, the expression: A LSS B AND C GEQ D is parsed as: A LSS (B AND C) GEQ D which is syntactically incorrect. The proper way of writting the expression is: (A LSS B) AND (C GEQ D)

It was indicated before that the right hand side of a register transfer operation

(←) must be an *arithmetic* expression. This does not allow the use of relational operators. In order to use them on the right hand side of a transfer, the (relational) expression must be enclosed in parenthesis. This in effect transforms the (relational) *expression* into a *term*, a valid *arithmetic-expression*, e.g.:

```
FLAG←(A NEQ B); ! Yields 0 or 1
TVAL←1+(D TST E); ! Yields 1,2, or 3
```

4. Register Transfer Sequences

The behavior of a digital system is described in ISPL by a list of statements. These statements can be build up from register transfers by using two special delimiters to indicate sequential or concurrent execution. Statement lists can be nested using parenthesis to build more complex statement lists. The syntax of the register transfer sequences is as follows:

statement-list *::=*	***parallel-statement-list*** |
	BAILOUT ***identifier*** |
	statement-list NEXT ***parallel-statement-list***
parallel-statement-list *::=*	***labelled-statement*** |
	parallel-statement-list ; ***labelled-statement***
labelled-statement *::=*	***statement*** |
	identifier := ***statement***
statement *::=*	***conditional-execute*** |
	conditional-decode |
	block |
	transfer |
	identifier
conditional-execute *::=*	(IF ***expression*** => ***statement-list***)
conditional-decode *::=*	(DECODE ***expression*** => ***parallel-statement-list***)
block *::=*	(***statement-list***)

4.1. Blocks

Blocks are the simplest building tools to define complicated statements. A ***block*** is a ***statement-list*** enclosed in parenthesis:

(A←0 NEXT A←A OR B[X]<7:0> ; C←C+1)

4.2. Conditional Statements

There are two ways of specifying conditional activities. These are the ***conditional-decode*** and the ***conditional-execute*** statements:

(condition => statement(s)),

where the conditions and their interpretation are as follows:

CONDITION	INTERPRETATION

DECODE ***expression*** The value of the expression is interpreted as an integer and used to select one out of n possible statements, given as a list of alternatives. These alternatives are separated by ";", but in this case they are not considered to be concurrent activities; only one of them will be executed. The statements in the list are numbered 0 through n-1, from left to right. The ith statement is executed if the value of the expression is equal to i.

IF ***expression*** This is a special case of the conditional-decode statement. The statement-list following the => operator is initiated if the logical value of the expression is TRUE, otherwise it is bypassed.

For simplicity, the ***expressions*** used in the ***conditional-execute*** statement do not have to be ***relational-expressions***, yielding a TRUE or FALSE value. An ***arithmetic-expression*** can be used, with the implication that the result of the expression is tested against 0. The ***statement-list*** is executed if the expression is not equal to 0, it is bypassed otherwise. In other words, the expression is interpreted as (expression NEQ 0). For similar reasons, the ***conditional-decode*** statement accepts a ***relation*** as the conditional expression, with the implication that the logical values FALSE and TRUE are interpreted as the numbers 0 and 1, respectively.

The language does not provide an IF ... THEN ... ELSE type of conditional statement. They are trivially described using a 2-way DECODE statement. The user should be careful to write the alternative statements in the proper order: the 0th case (logical FALSE) first and the 1st case (logical TRUE) second. Thus the statements are reversed from the normal Algol-like order.

Do not forget the ";"s after each alternative, except the last one, of a DECODE statement. A missing ";" in this context is a fatal error that is sometimes detected several lines after the offending alternative. The compiler will complain about a "missing action list".

4.3. Labelled Statements

The statements described above can be identified with a label. This label is used to designate the starting point of the statement. The label of a statement can be used wherever a statement is valid. The interpretation given to the use of a label in the middle of a *statement-list* is the following:

1) If the label is associated with a procedure definition, it is interpreted as a call (invocation) of the procedure, unless the invocation occurs inside the definition of the procedure, in which case the invocation is interpreted as a jump to the starting point of the sequence (i.e. there are no recursive calls in ISP).

2) Other invocations are treated as jumps to the starting point of the sequence. In the current implementation, labels (and their sequences) need not be declared before they are used. Thus we can jump forward in the description.

A reserved label, STOP, is predeclared in the compiler. It can be used to indicate a jump to the end of the description.

4.4. The BAILOUT Operation

The BAILOUT operation provides a way to describe the handling of exceptional conditions that might occur during the fetching, decoding, and execution of instructions. This operation is in effect a super RETURN from a procedure when an exceptional condition arises. The BAILOUT operator is used together with the label of the procedure whose context we want to leave, i.e., BAILOUT returns accross multiple levels of (dynamically) nested procedures. For instance:

Examples

```
p1 := (...NEXT (IF x => y←z NEXT BAILOUT p2) NEXT ...)

p2 := (...NEXT p1 NEXT ...)

Main := (...NEXT p2 NEXT ....)
```

In the above example, procedure MAIN invokes procedure P2 which starts execution of procedure P1. At some point, P1 decides that some error has occurred (IF X => ...) and that only MAIN can handle the situation. The effect of "BAILOUT P2" is to terminate the execution of P1 and P2 and return to procedure MAIN, at the point were it invoked P2.

4.5. Statement-Lists

Statements, labelled or otherwise, can be used to describe a list of concurrent activities, a ***parallel-statement-list***, using the ";" as delimiter. ***Parallel-statement-lists*** can be used to build sequences of activities or ***statement-lists***, using the "next" operator as delimiter. Notice that the ";" when used to indicate concurrency has a higher precedence than the "next" used to indicate sequentiality. For instance, in the following statement-list: A←B ; C←D NEXT E←F the transfers A←B and C←D are executed concurrently, and only when they are both completed will the locus of control pass to the next statement, the transfer E←F.

One detail to keep in mind is that ISPL is a statement language, not an expression language (in the BLISS sense). In particular, there is no such thing as an empty or null sequence, thus sequences like: (A←B;) or A←B; NEXT C←D are invalid (the ";" must be followed by a statement). In some cases the compiler is capable of detecting the extra ";" and will eliminate it after warning the user.

5. ISPL Programs

As mentioned in the Introduction, an ISPL description consists of a set of component declarations, together with a description of the behavior of the (main) system:

ispl-program ::= ***identifier*** := (***declaration-part statement-list***)

The above syntax indicates that ISPL programs look like labelled blocks, with a declaration-part, local to the body of the block.

EXAMPLE

```
MULT:=
        (DECLARE
        MPD<15:0>;
        P<15:0>;
        C<15:0>;
        STEP := (DECODE P<0> => P←P ↑SR 0; P←(P+MPD)<15:0> ↑SR 0)
         ERALCED
        L0:=    (
                C←8 NEXT
                L1:=    (
                        STEP NEXT
                        C←(C-1)<15:0> NEXT
                        (IF C NEQ 0 => L1)
                        )
                )
        )
```

The first example presents the ISPL description of a simple 8-bit multiplier using the shift-and-add algorithm. The multiplicand resides in the leftmost 8 bits of the MPD register. The multiplier resides in the rightmost 8 bits of the P register. The partial product is developed using all 16 bits of the P register. Additional details about the algorithm can be found in [Bell, 1972].

The description begins with the specification of the label for the program (MULTIPLIER). Labels are used in ISPL to identify activities so that they can be branched to, or used as subroutines.

The program itself is enclosed in parenthesis, and consists of two parts. The declarations and the specification of the behavior. The former are specified as a list of individual component declarations (multiplicand, multiplier/product, and step counter), and one procedure (STEP) which performs the basic multiplication operation, using the reserved identifiers DECLARE and ERALCED as brackets. The specification of the activities of the system is given as a list of two sequential steps. The first step (C←8) initialises the counter and the second is given by a labelled (L1) block of activities. this consists of a sequence of three steps. The first one performs the basic multiplication operation by calling the procedure; the second step decrements the counter; the third step tests the counter to see if the operation has been completed. If the value of the counter has not reached 0 then a jump to the label is indicated by using the label (L1) as an activity. If the counter is 0 then control flows out of the labelled statement and reaches the end of the program.

The basic multiplication operation is described using the DECODE control operation. It implements a 2-way branch depending on the value of the expression P<0>. The alternative paths selected by this operation are given as a list using the ";" as delimiter. The first path (P←P ↑SR 0) is selected if the value of the controlling expression (P<0>) is 0; the second path (P←(P+MPD) ↑SR 0) is selected if the value is 1. The operator ↑SR 0 represents a shift right inserting zero in the vacant position.

EXAMPLE

```
MINI:=(DECLARE  !MEMORY AND REGISTERS
        M[0:#377]<11:0>;        !MAIN MEMORY
        Z<7:0>; !EFFECTIVE ADDRESS REGISTER
        CACC<12:0>;     ! 13 BIT ACCUMULATOR WITH CARRY POSITION
                CARRY.BIT<> := CACC<12>;
                SIGN.BIT<> := CACC<11>;
                ACC<11:0> := CACC<11:0>;
        IR<11:0>;       !INSTRUCTION REGISTER
                OP<11:9> := IR<11:9>;
                I.BIT<> := IR<8>;
                ADDRESS<7:0> := IR<7:0>;
                IO.BITS<7:0> := IR<7:0>;
                UCLASS<> := IR<7>;
        L<7:0>; !RETURN REGISTER
        PC<7:0>;        !PROGRAM COUNTER
        IO.REG<7:0>;    !INPUT-OUTPUT REGISTER
        RUN<>;  !RUN MODE
    ! PROCEDURE TO INCREMENT PROGRAM COUNTER
    INCRPC:=( PC←(PC+1)<7:0>)   ! NOTE THAT PC WILL WRAP
  ERALCED
START:=     (DECODE RUN =>
                STOP;           ! If run=0
            (   IR←M[PC] NEXT INCRPC NEXT
                (DECODE I.BIT => Z←ADDRESS ; Z←M[ADDRESS]<7:0>) NEXT
                (DECODE OP =>   !INSTRUCTION DECODING
                        ACC←ACC AND M[Z];       !AND
                        CACC←ACC + M[Z];                !TAD (SETS CARRY BIT)
                        (M[Z]←(M[Z]+1)<11:0> NEXT (IF M[Z] EQL 0 => INCRPC) );  !ISZ
                        (M[Z]←ACC NEXT ACC←0);  !DCA
                        (L←PC NEXT PC←Z);               !JSR
                        PC←Z;                   !JUMP
                        IO.REG←IO.BITS;         !IOT
                        (DECODE UCLASS =>
                            (   (IF IR<6> => INCRPC) NEXT
                                (IF IR<5> => ACC← NOT ACC) NEXT
                                (IF IR<4> => ACC←0) NEXT
                                (IF IR<3> => CACC←ACC+1) NEXT   ! (SETS CARRY BIT)
                                (IF IR<2> => CACC←ACC-1) NEXT   ! (SETS CARRY BIT IF BORROW)
                                (IF IR<1> => ACC← ACC ↑SR0 1) NEXT
                                (IF IR<0> => ACC← ACC ↑SL0 1)   ); !END OF UCLASS=0
                            (   (IF IR<6> => INCRPC) NEXT
                                (IF IR<5> => PC←L) NEXT
                                (IF IR<4> => PC←CACC<7:0>) NEXT
                                (IF IR<3> => RUN←0) NEXT
                                (IF (IR<2> AND SIGN.BIT) OR
                                        (IR<1> AND (ACC EQL 0)) OR
                                        (IR<0> AND (NOT SIGN.BIT)) => INCRPC)
                            )
                        )       !END OF UCLASS DECODING
                )       !END OF INSTRUCTION DECODING
            )   !END OF RUN=1 MODE
            ) NEXT      !END OF INSTRUCTION CYCLE
        START
)
```

6. The Compiler Output

The compiler produces a listing file (with extension LST) and an "object code" file (with extension RTM). The latter extension stands for Register Transfer Machine. In other words, the compiler produces code for some idealized machine which executes register transfer operations.

6.1. Running the Compiler

The following example shows a typical execution. The actual calling procedure may change from installation to installation. When the compiler starts executing it prompts the user for the ISP source file name. If there are any error messages they are printed on the user's terminal as well as in the listing file. When the compilation is done (the compiler types messages indicating the current phase it is executing) it automatically calls the MACRO10 assembler and passes to it the name of the RTM file. At the end of the assembly the user should have the following files (assume the ISP source is called X.ISP): X.LST, X.RTM, X.REL, as well as the X.ISP file, of course.

```
ru isp
Input File: mult.isp

ISP COMPILER Thursday 29 Jul 76 23:42:13 MULT.ISP[N655M825]     PAGE 1

Parse Completed.
Optimization Completed.
Semantic Check and Output Follows
ISP:NO ERRORS DETECTED
23:43:57
MACRO:  .MAIN

EXIT
```

6.2. Example I - Listing

The listing file reproduces the ISPL source program together with any warning and error messages. The listing file is organized in 4 parts: 1) The listing proper, 2) A cross-reference listing indicating the places in the RTM object code were the registers, memories, and labels are being used, 3) A symbol table listing containing all the user and system declared entities, together with their attributes, and 4) A statement table listing containing a readable version of the RTM object code.

```
[001] MULT:=
[001]   (DECLARE
[002]  MPD<15:0>;
[002]  P<15:0>;
[002]  C<15:0>;
[002]  STEP := (DECODE P<0> => P←P ↑SR 0; P←(P+MPD)<15:0> ↑SR 0)
[003]   ERALCED
[003]  L0:=    (
[003]          C←8 NEXT
[004]          L1:=    (
[004]                  STEP NEXT
[005]                  C←(C-1)<15:0> NEXT
[005]                  (IF C NEQ 0 => L1)
[005]                  )
[005]          )
[005]  )
[005]
```

6.3. Example I - Symbol Table

The compiler produced symbol table for the multiplier example is shown below. There is an entry (1 line) for each user or compiler declared component. These include memory components, labels, and constants. The INDEX column indicates the position in the symbol table of the entity. This index is used to represent the variables in the statement table.

```
ISP COMPILER Thursday 29 Jul 76 22:09:14 TEMP.TMP[N655MB25]     PAGE 2

   0    0 10000000    0    0    0    0        0 'e     '
   1    2 10000000    0    0    0   20        1 'C     '<0(17):17(0)>
   2    4 10000100    0    0   16    0        0 'L0    '
   3    4 10000100    0    0   21    0        0 'L1    '
```

```
 4    2 10000000   0    0    0   20        1 'MPD    '<0(17):17(0)>
 5    4 10000100   0    0    1    0        0 'MULT   '
 6    2 10000000   0    0    0   20        1 'P      '<0(17):17(0)>
 7    4 10001100   0    0    3    0        0 'STEP   '
10    4 10000101   0    0   35    0        0 'STOP   '
11    3 10000001   0    0    0    1        0         0
12    5 10000001   0    0    0    1        0     0,, 0
13    3 10000001   0    0    0    1        0         1
14    3 10000001   0    0    0    4        0        10
15    5 10000001   0    0    0   20        0   17,, 0
16   10 10000001   0    0    0    1        0 '%TFAAA'
17    7 10000001   0    0    0   21        0 '%TRAAA'
20    7 10000001   0    0    0   20        0 '%TRAAB'
21    7 10000001   0    0    0    1        0 '%TRAAC'
```

The TYPE column describes the type of "variable" stored in a given entry of the symbol table. The valid types are: Memory Array (TYPE=1), Register (2), Constant (3), Label (4), Mask (5), Flag (6), Temporary register (7), and Temporary flag (10). The last two are used for compiler declared variables (for instance, temporary registers are declared in order to store partial results when evaluating expressions).

The FLAGS field contains information used by the compiler. It is displayed as part of the output mainly for debugging purposses (i.e. they show the status of the symbol table entry).

The DEF field is used to store a pointer to an associated symbol table entry. It is used when a memory component, say a register, is defined in terms of a previously declared memory component. For instance, we can declare:

```
INSTRUCTION.REGISTER<15:0>;

OP.CODE<3:0> := INSTRUCTION.REGISTER<15:12>;
```

In the symbol table listing, the DEF field for OP.CODE will point to a pseudo register declaration entry, corresponding to INSTRUCTION.REGISTER<15:12>. The DEF field for the latter will point to the main declaration of INSTRUCTION.REGISTER<15:0>. If INSTRUCTION.REGISTER had been mapped on top of another register or memory

declaration, the DEF fields will chain these definitions. (DEF defines a chain of definitions, the last entry of which is always the main declaration).

The LBL (LaBeL) field associates with every user declared label, an integer used by the compiler. This integer constitutes an internal label.

The BCNT and WCNT (Bit CouNT and Word CouNT, respectively) indicate the number of bits and words for each memory and constant. (The count is given as an octal number).

The PNAME (Print NAME) contains an identifier for each entry. For user declared variables and labels it contains the identifier used in the program (truncated to six characters). Constants are identified by their numeric value (octal). Masks are represented as a pair of octal numbers. These indicate the left and rightmost bit positions of the mask with respect to the right edge of the word (for instance, a binary mask like 00011000 will appear as 4,,3). System declared registers and flags are given compiler generated names.

The last field of the symbol table, WORDS;BITS, contains the list of subcomponents for each user declared memory or register. The list contains the bit (word) names given in the declaration as well as the internal bit (word) names generated and used for the compiler. The compiler generates a position dependent internal bit (word) name which can be used to generate the proper subcomponent accessing code. These position identifiers are indicated in parenthesis, next to the user specified bit (or word) names.

6.4. Example I - Cross Reference

```
INDEX   VAR      STATEMENTS

    1   'C      '
           20        24        25        26
    2   'L0     '
           33
    3   'L1     '
           30        32
    4   'MPD    '
           11
    5   'MULT   '
           34
    6   'P      '
            5         7        11        13
    7   'STEP   '
           15        23
   10   'STOP   '
   11           0
            7        13        26
   12       0,, 0
            5
   13           1
   14          10
           20
   15       17,, 0
           12        25
   16   '%TFAAA'
           26        27
   17   '%TRAAA'
           11        12        24        25
   20   '%TRAAB'
           12        13
   21   '%TRAAC'
            5         6
```

6.5. Example 1 - Statement Table

```
INDEX  LABEL    FLAG    OPCODE  DEST    SOURCE1 SOURCE2 MERGE   PATHS

    0              0    'START '                          16
    1  'MULT  '    1    'SMERGE'
       (     5)
    2              0    'ISP   '                           2
    3  'STEP  '    1    'SMERGE'
       (     7)
    4              0    'ISP   '                           3
    5              0    'RBYTE ''%TRAAC''P     '   0,, 0
                                (    21)(     6)(    12)
    6              0    'BRANCH'        '%TRAAC'          14  7,11
                                (    21)
    7              0    'RSHFT ''P     ''P     '       0
                                (     6)(     6)(    11)
   10              0    'JOIN  '                          14
   11              0    'ADD   ''%TRAAA''P     ''MPD   '
                                (    17)(     6)(     4)
   12              0    'RBYTE ''%TRAAB''%TRAAA'  17,, 0
                                (    20)(    17)(    15)
   13              0    'RSHFT ''P     ''%TRAAB'       0
                                (     6)(    20)(    11)
   14              0    'SMERGE'
   15              0    'RETURN'        'STEP  '           3
                                (     7)
   16  'L0    '    1    'SMERGE'
       (     2)
   17              0    'ISP   '                           4
   20              0    'MOVE  ''C     '      10
                                (     1)(    14)
   21  'L1    '    1    'SMERGE'
       (     3)
   22              0    'ISP   '                           5
   23              0    'CALL  '        'STEP  '           3
                                (     7)
   24              0    'DECR  ''%TRAAA''C     '
                                (    17)(     1)
   25              0    'RBYTE ''C     ''%TRAAA'  17,, 0
                                (     1)(    17)(    15)
   26              0    'NEQ   ''%TFAAA''C     '       0
                                (    16)(     1)(    11)
   27              0    'IF    '        '%TFAAA'          31  31,30
                                (    16)
   30              4    'JOIN  '        'L1    '          21
                                (     3)
   31              0    'SMERGE'
   32              0    'NOOP  '        'L1    '          21
                                (     3)
   33              0    'NOOP  '        'L0    '          16
                                (     2)
   34              0    'NOOP  '        'MULT  '           1
                                (     5)
   35  'STOP  '    1    'STOP  '
       (    10)
```

The LABEL field is used to identify the individual statements.

The FLAGS field, as in the symbol table, is used internally by the compiler. In this particular example, the only flag shown indicates whether the label associated with the instruction was declared by the user (1) or by the compiler(0).

The OPR field contains the operation name. The meaning of most operations should be obvious from their names. Data operations are described as a 3-address assembly-like instruction. The source operands and the destination operand are indicated by their index into the symbol table (columns SRC1, SRC2, and DEST). The RBYTE operation is used to extract a byte from a register. The interpretation of the operation is the following: DESTINATION←SOURCE1<SOURCE2> where destination and source1 are of type register and source2 is a mask. Other non-obvious data operations (not shown in the example) are:

WBYTE (DESTINATION<SOURCE1>←SOURCE2),

READ (DESTINATION←SOURCE1[SOURCE2]), and

WRITE (DESTINATION[SOURCE1]←SOURCE2).

The RTM code uses at most three operands, thus an ISP statement like: A←B[C]<1> compiles into two RTM operations. The first is a READ operation that loads a (compiler generated) temporary register with B[C]. The second operation is a RBYTE that extracts bit 1 of this temporary (the position of this bit is deduced from the declaration of B) and stores it into A. Control operations are slightly more complex. Serial Merge (SMERGEOP) operations are used as merging points for non-concurrent sequences. Parallel merge (PMERGEOP) operations are used as merging points for concurrent sequences. Branch (BRANCHOP) operators select one out of many alternative sequences. These sequences are identified by a list of the labels of their

entry points, given in the same order as the conditional statement in the original ISP. Diverge (DIVERGEOP) operations are used to initiate simultaneous, concurrent paths. These paths are, as in the branch operations, indicated by a list of labels.

Branch and Diverge operations also specify the label of the statement following the alternative or concurrent paths. That statement is the "merge" point for the different paths.

The join (JOIN) operator is used as an unconditional jump statement. It generally appears as the last statement of a path, and jumps to the appropriate merging point (a serial or parallel merge). The NOOP operation is used as a control operation. It is generated by the compiler to indicate the end of a block. The statement points to the entry point of the block.

7. References

[Barbacci, 1973] Barbacci, M. R. and D. P. Siewiorek: "The Automated Exploration of the Design Space for Register Transfer (RT) Systems". First Annual Symposium on Computer Architecture, University of Florida, Gainesville, Florida, December 1973.

[Bell, 1971] Bell, C. G. and A. Newell: "Computer Structures: Readings and Examples". McGraw Hill Book Company, New York, 1971.

[Bell, 1972] Bell, C. G., J. Grason, and A. Newell: "Designing Computers and Digital Systems". Digital Press, Digital Equipment Corporation, 1972.

8. Appendix I - The Minicomputer Listing

```
[001] MINI:=(DECLARE    !MEMORY AND REGISTERS
[002]   M[0:#377]<11:0>;        !MAIN MEMORY
[002]   Z<7:0>; !EFFECTIVE ADDRESS REGISTER
[002]   CACC<12:0>;     ! 13 BIT ACCUMULATOR WITH CARRY POSITION
[002]           CARRY.BIT<> := CACC<12>;
[002]           SIGN.BIT<> := CACC<11>;
[002]           ACC<11:0> := CACC<11:0>;
[002]   IR<11:0>;       !INSTRUCTION REGISTER
[002]           OP<11:9> := IR<11:9>;
[002]           I.BIT<> := IR<8>;
[002]           ADDRESS<7:0> := IR<7:0>;
[002]           IO.BITS<7:0> := IR<7:0>;
[002]           UCLASS<> := IR<7>;
[002]   L<7:0>; !RETURN REGISTER
[002]   PC<7:0>;        !PROGRAM COUNTER
[002]   IO.REG<7:0>;    !INPUT-OUTPUT REGISTER
[002]   RUN<>;  !RUN MODE
[002]     ! PROCEDURE TO INCREMENT PROGRAM COUNTER
[002]     INCRPC:=( PC←(PC+1)<7:0>)     ! NOTE THAT PC WILL WRAP
[003]   ERALCED
[003] START:=       (DECODE RUN =>
[004]           STOP;           ! If run=0
[004]       (   IR←M[PC] NEXT INCRPC NEXT
[004]           (DECODE I.BIT => Z←ADDRESS ; Z←M[ADDRESS]<7:0>) NEXT
[004]           (DECODE OP =>   !INSTRUCTION DECODING
[004]                   ACC←ACC AND M[Z];       !AND
[004]                   CACC←ACC + M[Z];                !TAD (SETS CARRY BIT)
[004]                   (M[Z]←(M[Z]+1)<11:0> NEXT (IF M[Z] EQL 0 => INCRPC) );  !ISZ
[004]                   (M[Z]←ACC NEXT ACC←0);  !DCA
[004]                   (L←PC NEXT PC←Z);               !JSR
[004]                   PC←Z;                   !JUMP
[004]                   IO.REG←IO.BITS;         !IOT
[004]                   (DECODE UCLASS =>
[004]                       (   (IF IR<6> => INCRPC) NEXT
[004]                           (IF IR<5> => ACC← NOT ACC) NEXT
[004]                           (IF IR<4> => ACC←0) NEXT
[004]                           (IF IR<3> => CACC←ACC+1) NEXT   ! (SETS CARRY BIT)
[004]                           (IF IR<2> => CACC←ACC-1) NEXT   ! (SETS CARRY BIT IF BORROW)
[004]                           (IF IR<1> => ACC← ACC ↑SR0 1) NEXT
[004]                           (IF IR<0> => ACC← ACC ↑SL0 1)   ); !END OF UCLASS=0
[004]                       (   (IF IR<6> => INCRPC) NEXT
[004]                           (IF IR<5> => PC←L) NEXT
[004]                           (IF IR<4> => PC←CACC<7:0>) NEXT
[004]                           (IF IR<3> => RUN←0) NEXT
[004]                           (IF (IR<2> AND SIGN.BIT) OR
[004]                                   (IR<1> AND (ACC EQL 0)) OR
[004]                                   (IR<0> AND (NOT SIGN.BIT)) => INCRPC)
[004]                       )
[004]                   )       !END OF UCLASS DECODING
[004]           )       !END OF INSTRUCTION DECODING
[004]       )   !END OF RUN=1 MODE
[004]       ) NEXT      !END OF INSTRUCTION CYCLE
[004]   START
[004] )
```

INDEX	TYPE	FLAGS	DEF	BLK	LBL	BCNT	WCNT	PNAME	WORDS;BITS:NAME(POSITION)
0	0	10000000	0	0	0	0	0	'e	'
1	2	10010000	4	0	0	14	1	'ACC	'<0(13):13(0)>
2	2	10010000	16	0	0	10	1	'ADDRES'	<0(7):7(0)>
3	2	10100000	5	0	0	1	1	'CACC	'<13(0)>
4	2	10100000	5	0	0	14	1	'CACC	'<0(13):13(0)>
5	2	10000000	0	0	0	15	1	'CACC	'<0(14):14(0)>
6	2	10100000	5	0	0	1	1	'CACC	'<14(0)>
7	2	10010000	6	0	0	1	1	'CARRY.'	
10	2	10010000	15	0	0	1	1	'I.BIT '	
11	4	10001100	0	0	3	0	0	'INCRPC'	
12	2	10010000	20	0	0	10	1	'IO.BIT'	<0(7):7(0)>
13	2	10000000	0	0	0	10	1	'IO.REG'	<0(7):7(0)>
14	2	10100000	17	0	0	3	1	'IR	'<11(2):13(0)>
15	2	10100000	17	0	0	1	1	'IR	'<10(0)>
16	2	10100000	17	0	0	10	1	'IR	'<0(7):7(0)>
17	2	10000000	0	0	0	14	1	'IR	'<0(13):13(0)>
20	2	10100000	17	0	0	10	1	'IR	'<0(7):7(0)>
21	2	10100000	17	0	0	1	1	'IR	'<7(0)>
22	2	10000000	0	0	0	10	1	'L	'<0(7):7(0)>
23	1	10000000	0	0	0	14	400	'M	'[377(377):0(0)]<0(13):13(0)>
24	4	10000100	0	0	1	0	0	'MINI	'
25	2	10010000	14	0	0	3	1	'OP	'<11(2):13(0)>
26	2	10000000	0	0	0	10	1	'PC	'<0(7):7(0)>
27	2	10000000	0	0	0	1	1	'RUN	'
30	2	10010000	3	0	0	1	1	'SIGN.B'	
31	4	10000100	0	0	10	0	0	'START '	
32	4	10000101	0	0	161	0	0	'STOP	'
33	2	10010000	21	0	0	1	1	'UCLASS'	
34	2	10000000	0	0	0	10	1	'Z	'<0(7):7(0)>
35	5	10000001	0	0	0	1	0	0,, 0	
36	3	10000001	0	0	0	1	0	0	
37	3	10000001	0	0	0	1	0	1	
40	5	10000001	0	0	0	1	0	1., 1	
41	5	10000001	0	0	0	1	0	2,, 2	
42	5	10000001	0	0	0	1	0	3,, 3	
43	5	10000001	0	0	0	1	0	4,, 4	
44	5	10000001	0	0	0	1	0	5,, 5	
45	5	10000001	0	0	0	1	0	6,, 6	
46	5	10000001	0	0	0	10	0	7,, 0	
47	5	10000001	0	0	0	14	0	13,, 0	
50	10	10000001	0	0	0	1	0	'XTFAAA'	
51	7	10000001	0	0	0	14	0	'XTRAAA'	
52	7	10000001	0	0	0	15	0	'XTRAAB'	
53	7	10000001	0	0	0	14	0	'XTRAAC'	
54	7	10000001	0	0	0	1	0	'XTRAAD'	
55	7	10000001	0	0	0	1	0	'XTRAAE'	
56	7	10000001	0	0	0	1	0	'XTRAAF'	
57	7	10000001	0	0	0	11	0	'XTRAAG'	
60	7	10000001	0	0	0	1	0	'XTRAAH'	

```
INDEX   VAR      STATEMENTS

    1   'ACC   '
          27       32      46      47      67      73      77     103
                  107     113     140
    2   'ADDRES'
          20       22
    5   'CACC  '
          32       77     103     130
    7   'CARRY.'
   10   'I.BIT '
          17
   11   'INCRPC'
           7       16      43      63     120     151
   12   'IO.BIT'
          56
   13   'IO.REG'
          56
   17   'IR    '
          15       61      65      71      75     101     105     111
                  116     122     126     132     136     141     145
   22   'L     '
          51      124
   23   'M     '
          15       22      26      31      34      37      40      46
   24   'MINI  '
         160
   25   'OP    '
          25
   26   'PC    '
           5        6      15      51      52      54     124     130
   27   'RUN   '
          12      134
   30   'SIGN.B'
         137      144
   31   'START '
         156      157
   32   'STOP  '
          13
   33   'UCLASS'
          60
   34   'Z     '
          20       23      26      31      34      37      40      46
                   52      54
   35     0,, 0
         111      145
   36         0
          41      140
   37         1
         107      113
   40     1,, 1
```

```
        105     141
41       2,, 2
        101     136
42       3,, 3
         75     132
43       4,, 4
         71     126
44       5,, 5
         65     122
45       6,, 6
         61     116
46       7,, 0
          6      23     130
47      13,, 0
         36
50    '%TFAAA'
         41      42     140     142
51    '%TRAAA'
         22      23      26      27      31      32      34      35
                 40      41
52    '%TRAAB'
         35      36
53    '%TRAAC'
         36      37
54    '%TRAAD'
         61      62      65      66      71      72      75      76
                101     102     105     106     111     112     116     117
                122     123     126     127     132     133     136     137
                143     147     150
55    '%TRAAE'
        141     142     143
56    '%TRAAF'
        144     146
57    '%TRAAG'
          5       6
60    '%TRAAH'
        145     146     147
```

```
INDEX  LABEL    FLAG    OPCODE  DEST    SOURCE1 SOURCE2 MERGE   PATHS

    0              0    'START '                          10
    1  'MINI  '    1    'SMERGE'
       (    24)
    2              0    'ISP   '                           2
    3  'INCRPC'    1    'SMERGE'
       (    11)
    4              0    'ISP   '                           3
    5              0    'INCR  ''%TRAAG''PC    '
                              (    57)(    26)
    6              0    'RBYTE ''PC    ''%TRAAG'   7,, 0
                              (    26)(    57)(    46)
    7              0    'RETURN'        'INCRPC'           3
                                        (    11)
   10  'START '    1    'SMERGE'
       (    31)
   11              0    'ISP   '                           4
   12              0    'BRANCH'        'RUN   '         155  13,15
                                        (    27)
   13              0    'JOIN  '        'STOP  '         161
                                        (    32)
   14              0    'JOIN  '                         155
   15              0    'READ  ''IR    ''M     ''PC    '
                              (    17)(    23)(    26)
   16              0    'CALL  '        'INCRPC'           3
                                        (    11)
   17              0    'BRANCH'        'I.BIT '          24  20,22
                                        (    10)
   20              0    'MOVE  ''Z     ''ADDRES'
                              (    34)(     2)
   21              0    'JOIN  '                          24
   22              0    'READ  ''%TRAAA''M     ''ADDRES'
                              (    51)(    23)(     2)
   23              0    'RBYTE ''Z     ''%TRAAA'   7,, 0
                              (    34)(    51)(    46)
   24              0    'SMERGE'
   25              0    'BRANCH'        'OP    '         154  26,31,34,46,51,54,56,
          60
                                        (    25)
   26              0    'READ  ''%TRAAA''M     ''Z     '
                              (    51)(    23)(    34)
   27              0    'AND   ''ACC   ''ACC   ''%TRAAA'
                              (     1)(     1)(    51)
   30              0    'JOIN  '                         154
   31              0    'READ  ''%TRAAA''M     ''Z     '
                              (    51)(    23)(    34)
   32              0    'ADD   ''CACC  ''ACC   ''%TRAAA'
                              (     5)(     1)(    51)
   33              0    'JOIN  '                         154
   34              0    'READ  ''%TRAAA''M     ''Z     '
                              (    51)(    23)(    34)
   35              0    'INCR  ''%TRAAB''%TRAAA'
                              (    52)(    51)
   36              0    'RBYTE ''%TRAAC''%TRAAB'  13,, 0
```

```
                        (    53)(    52)(    47)
37          0    'WRITE ''M     ''Z     ''%TRAAC'
                        (    23)(    34)(    53)
40          0    'READ  ''%TRAAA''M     ''Z     '
                        (    51)(    23)(    34)
41          0    'EQL   ''%TFAAA''%TRAAA'       0
                        (    50)(    51)(    36)
42          0    'IF    '         '%TFAAA'            44  44,43
                                   (    50)
43          0    'CALL  '         'INCRPC'             3
                                   (    11)
44          0    'SMERGE'
45          0    'JOIN  '                            154
46          0    'WRITE ''M     ''Z     ''ACC   '
                        (    23)(    34)(     1)
47          0    'CLEAR ''ACC   '
                        (     1)
50          0    'JOIN  '                            154
51          0    'MOVE  ''L     ''PC    '
                        (    22)(    26)
52          0    'MOVE  ''PC    ''Z     '
                        (    26)(    34)
53          0    'JOIN  '                            154
54          0    'MOVE  ''PC    ''Z     '
                        (    26)(    34)
55          0    'JOIN  '                            154
56          0    'MOVE  ''IO.REG''IO.BIT'
                        (    13)(    12)
57          0    'JOIN  '                            154
60          0    'BRANCH'         'UCLASS'            153  61,116
                                   (    33)
61          0    'RBYTE ''%TRAAD''IR    '    6,, 6
                        (    54)(    17)(    45)
62          0    'IF    '         '%TRAAD'            64  64,63
                                   (    54)
63          0    'CALL  '         'INCRPC'             3
                                   (    11)
64          0    'SMERGE'
65          0    'RBYTE ''%TRAAD''IR    '    5,, 5
                        (    54)(    17)(    44)
66          0    'IF    '         '%TRAAD'            70  70,67
                                   (    54)
67          0    'NOT   ''ACC   ''ACC   '
                        (     1)(     1)
70          0    'SMERGE'
71          0    'RBYTE ''%TRAAD''IR    '    4,, 4
                        (    54)(    17)(    43)
72          0    'IF    '         '%TRAAD'            74  74,73
                                   (    54)
73          0    'CLEAR ''ACC   '
                        (     1)
74          0    'SMERGE'
75          0    'RBYTE ''%TRAAD''IR    '    3,, 3
                        (    54)(    17)(    42)
```

```
INDEX  LABEL   FLAG   OPCODE  DEST    SOURCE1 SOURCE2 MERGE   PATHS

  76             0    'IF    '        '%TRAAD'          100  100,77
                                      (   54)
  77             0    'INCR  ''CACC  ''ACC   '
                       (    5)(    1)
 100             0    'SMERGE'
 101             0    'RBYTE ''%TRAAD''IR    '   2,, 2
                       (   54)(   17)(   41)
 102             0    'IF    '        '%TRAAD'          104  104,103
                                      (   54)
 103             0    'DECR  ''CACC  ''ACC   '
                       (    5)(    1)
 104             0    'SMERGE'
 105             0    'RBYTE ''%TRAAD''IR    '   1,, 1
                       (   54)(   17)(   40)
 106             0    'IF    '        '%TRAAD'          110  110,107
                                      (   54)
 107             0    'RSHFT0''ACC   ''ACC   '      1
                       (    1)(    1)(   37)
 110             0    'SMERGE'
 111             0    'RBYTE ''%TRAAD''IR    '   0,, 0
                       (   54)(   17)(   35)
 112             0    'IF    '        '%TRAAD'          114  114,113
                                      (   54)
 113             0    'LSHFT0''ACC   ''ACC   '      1
                       (    1)(    1)(   37)
 114             0    'SMERGE'
 115             0    'JOIN  '                        153
 116             0    'RBYTE ''%TRAAD''IR    '   6,, 6
                       (   54)(   17)(   45)
 117             0    'IF    '        '%TRAAD'          121  121,120
                                      (   54)
 120             0    'CALL  '        'INCRPC'          3
                                      (   11)
 121             0    'SMERGE'
 122             0    'RBYTE ''%TRAAD''IR    '   5,, 5
                       (   54)(   17)(   44)
 123             0    'IF    '        '%TRAAD'          125  125,124
                                      (   54)
 124             0    'MOVE  ''PC    ''L     '
                       (   26)(   22)
 125             0    'SMERGE'
 126             0    'RBYTE ''%TRAAD''IR    '   4,, 4
                       (   54)(   17)(   43)
 127             0    'IF    '        '%TRAAD'          131  131,130
                                      (   54)
 130             0    'RBYTE ''PC    ''CACC  '   7,, 0
                       (   26)(    5)(   46)
 131             0    'SMERGE'
 132             0    'RBYTE ''%TRAAD''IR    '   3,, 3
                       (   54)(   17)(   42)
 133             0    'IF    '        '%TRAAD'          135  135,134
                                      (   54)
 134             0    'CLEAR ''RUN   '
                       (   27)
```

```
INDEX  LABEL    FLAG    OPCODE  DEST    SOURCE1 SOURCE2 MERGE   PATHS

  135               0    'SMERGE'
  136               0    'RBYTE ''%TRAAD''IR    '   2,, 2
                              (    54)(    17)(    41)
  137               0    'AND   ''%TRAAD''%TRAAD''SIGN.B'
                              (    54)(    54)(    30)
  140               0    'EQL   ''%TFAAA''ACC   '       0
                              (    50)(     1)(    36)
  141               0    'RBYTE ''%TRAAE''IR    '   1,, 1
                              (    55)(    17)(    40)
  142               0    'AND   ''%TRAAE''%TRAAE''%TFAAA'
                              (    55)(    55)(    50)
  143               0    'OR    ''%TRAAD''%TRAAD''%TRAAE'
                              (    54)(    54)(    55)
  144               0    'NOT   ''%TRAAF''SIGN.B'
                              (    56)(    30)
  145               0    'RBYTE ''%TRAAH''IR    '   0,, 0
                              (    60)(    17)(    35)
  146               0    'AND   ''%TRAAH''%TRAAH''%TRAAF'
                              (    60)(    60)(    56)
  147               0    'OR    ''%TRAAD''%TRAAD''%TRAAH'
                              (    54)(    54)(    60)
  150               0    'IF    '        '%TRAAD'          152  152,151
                                         (    54)
  151               0    'CALL  '        'INCRPC'           3
                                         (    11)
  152               0    'SMERGE'
  153               0    'SMERGE'
  154               0    'SMERGE'
  155               0    'SMERGE'
  156               0    'NOOP  '        'START '           10
                                         (    31)
  157               0    'JOIN  '        'START '           10
                                         (    31)
  160               0    'NOOP  '        'MINI  '           1
                                         (    24)
  161  'STOP  '     1    'STOP  '
       (    32)
```

9. Appendix II - ISPL Reserved Keywords

The following keywords and identifiers are reserved in the language:

AND
BAILOUT
DECLARE
DECODE
DELAY (not described in this manual)
EQL
EQV
ERALCED
GEQ
GTR
IF
LSS
LEQ
MACRO
MINUS
NEQ
NEXT
NOT
OR
STOP
TST
WAIT (not described in this manual)
XOR

10. Appendix III - The XTOP10.REQ File

```
XTTESTOP=#200,
XTEQLOP=#201,
XTNEQOP=#202,
XTLSSOP=#203,
XTLEQOP=#204,
XTGEQOP=#205,
XTGTROP=#206,
XTMOVEOP=#210,
XTCLEAROP=#211,
XTNOOP=#212,
XTWBYTEOP=#213,
XTRBYTEOP=#214,
XTREADOP=#220,
XTWRITEOP=#221,
XTLROTOP=#226,
XTRROTOP=#227,
XTNOTOP=#230,
XTINCROP=#231,
XTDECROP=#232,
XTLSHFTOP=#233,
XTRSHFTOP=#234,
XTANDOP=#235,
XTOROP=#236,
XTXOROP=#241,
XTEQVOP=#242,
XTADDOP=#243,
XTSUBOP=#244,
XTLSHFT1OP=#245,
XTRSHFT1OP=#246,
XTLSHFT8OP=#247,
XTRSHFT8OP=#250,
XTCONCOP=#251,
XTNEGOP=#252,
XTSUBTWOOP=#253,        !Two's Complement Subtract
XTMULTOP=#300,
XTDIVOP=#301,
XTIFOP=#350,
XTRETURNOP=#351,
XTISPOP=#352,
XTPJOINOP=#353,
XTBAILOUTOP=#361,
XTCALLOP=#363,
XTJOINOP=#365,
XTBRANCHOP=#371,
XTDIVERGEOP=#372,
XTSMERGEOP=#373,
XTPMERGEOP=#374,
XTSTARTOP=#376,
XTSTOPOP=#377,
```

11. Appendix IV - The Multiplier MACRO10 Format

Another version of the RTM code intended for machine consumption consists of a MACRO10 program in which all the information in the symbol and statement tables is encoded as MACRO10 statements (all of which are in fact, data definition statements).

In order to understand the RTM file (the ISP and listing files associated with this example were described previously, in the section describing the compiler output), the reader should have a working knowledge of BLISS10, enough to understand the SIMISP.REQ file describing the structure of the MACRO10 statements. The SIMISP.REQ file is given after the example.

```
;ARF ISP COMPILER - JUNE 1976
        TWOSEG
        INTERN  SYTABL,STTABL,SYTOP,STTOP,ISPTIT
        INTERN  ISPFNM,ISPEXT,ISPDAT,ISPTIM,ISPPPN,ISPVER
        RELOC 400000
%B0005: EXP     0,17,17,0,-1
%B0003: EXP     0,17,17,0,-1
%B0001: EXP     0,17,17,0,-1
$00025: EXP     27,26
$00006: EXP     7,11
        RELOC   0
SYTABL:
        BYTE    (9)0,200(18)0,0,0,0,0(36)'@     ',0
        BYTE    (9)2,200(18)0,0,20,0,%B0001(36)'C     ',1
        BYTE    (9)4,204(18)0,17,0,0,0(36)'L1    ',0
        BYTE    (9)2,200(18)0,0,20,0,%B0003(36)'MPD   ',1
        BYTE    (9)4,204(18)0,1,0,0,0(36)'MULT  ',0
        BYTE    (9)2,200(18)0,0,20,0,%B0005(36)'P     ',1
        BYTE    (9)4,214(18)0,3,0,0,0(36)'STEP  ',0
        BYTE    (9)4,205(18)0,32,0,0,0(36)'STOP  ',0
        BYTE    (9)5,201(18)0,0,1,0,0(36)       0,0
        BYTE    (9)3,201(18)0,0,1,0,0(36)       0,0
        BYTE    (9)3,201(18)0,0,1,0,0(36)       1,0
        BYTE    (9)3,201(18)0,0,4,0,0(36)      10,0
        BYTE    (9)5,201(18)0,0,20,0,0(36)17000000,0
        BYTE    (9)10,201(18)0,0,1,0,0(36)'%TFAAA',0
        BYTE    (9)7,201(18)0,0,21,0,0(36)'%TRAAA',0
        BYTE    (9)7,201(18)0,0,20,0,0(36)'%TRAAB',0
        BYTE    (9)7,201(18)0,0,1,0,0(36)'%TRAAC',0
STTABL:
        BYTE    (9)0,376(18)2361(12)0,0,0(18)0,16,0,0
        BYTE    (9)1,373(18)5261(12)0,0,0(18)0,0,0,4
        BYTE    (9)0,352(18)4301(12)0,0,0(18)0,2,0,0
        BYTE    (9)1,373(18)5261(12)0,0,0(18)0,0,0,6
        BYTE    (9)0,352(18)4301(12)0,0,0(18)0,3,0,0
```

```
        BYTE    (9)0,214(18)11221(12)20,5,10(18)0,0,0,0
        BYTE    (9)0,371(18)13461(12)0,20,0(18)2,14,$00006,0
        BYTE    (9)0,234(18)7070(12)5,5,11(18)0,0,0,0
        BYTE    (9)0,365(18)2341(12)0,0,0(18)0,14,0,0
        BYTE    (9)0,243(18)7121(12)16,5,3(18)0,0,0,0
        BYTE    (9)0,214(18)11221(12)17,16,14(18)0,0,0,0
        BYTE    (9)0,234(18)7070(12)5,17,11(18)0,0,0,0
        BYTE    (9)0,373(18)5261(12)0,0,0(18)0,0,0,0
        BYTE    (9)0,351(18)1421(12)0,6,0(18)0,3,0,0
        BYTE    (9)0,210(18)10021(12)1,13,0(18)0,0,0,0
        BYTE    (9)1,373(18)5261(12)0,0,0(18)0,0,0,2
        BYTE    (9)0,352(18)4301(12)0,0,0(18)0,4,0,0
        BYTE    (9)0,363(18)2401(12)0,6,0(18)0,3,0,0
        BYTE    (9)0,232(18)10024(12)16,1,0(18)0,0,0,0
        BYTE    (9)0,214(18)11221(12)1,16,14(18)0,0,0,0
        BYTE    (9)0,202(18)7042(12)15,1,11(18)0,0,0,0
        BYTE    (9)0,350(18)6241(12)0,15,0(18)2,27,$00025,0
        BYTE    (9)4,365(18)2341(12)0,2,0(18)0,17,0,0
        BYTE    (9)0,373(18)5261(12)0,0,0(18)0,0,0,0
        BYTE    (9)0,212(18)212(12)0,2,0(18)0,17,0,0
        BYTE    (9)0,212(18)212(12)0,4,0(18)0,1,0,0
        BYTE    (9)1,377(18)1441(12)0,0,0(18)0,0,0,7
SYTOP:  EXP     20
STTOP:  EXP     32
ISPTIT: ASCIZ   'Friday 23 Jul 76 19:22:58 TEST.ISP[N655MB25]'
ISPFNM: SIXBIT  'TEST  '
ISPEXT: SIXBIT  'ISP   '
ISPDAT: ASCIZ   '23 Jul 76'
ISPTIM: ASCIZ   '19:22:58'
ISPPPN: EXP     32540,334165
ISPVER: EXP     0,0,0,0
        END
```

The MACRO10 program starts by declaring certain symbols to be accessible to separately compiled modules. This is done with the INTERN MACRO10 operator. The symbols in question are the base address for the symbol and statement tables and the number of entries in each table (actually the index of the last entry, the first entry has index 0). The user therefore can access the symbol table entries between SYTABL[0,<fieldname>] and SYTABL[@SYTOP,<fieldname>] and the statement table entries between STTABL[0,<fieldname>] and STTABL[@STTOP,<fieldname>].

The MACRO10 program is divided in two segments, the high segment contains the bit and word lists of the symbol table, as well as the label lists of the statement table. The low segment contains the symbol and statement tables properly.

The bit and word lists are declared as a list of expressions, using the EXP MACRO10 operation, each element of the list takes a full word on the PDP-10. Each bit and word list is identified by a label of the form: %Bnnnn for bit lists and %Wnnnn for word lists were nnnn is the index of the symbol table associated with the bit/word list. Every element of a bit/word list appears as a pair of consecutive elements in the EXP statement. The first (odd) element is the bit/word name. The second (even) element is the bit/word position. The bit/word list ends with a -1 as a bit/word name element.

The statement table label lists appear as lists of expressions, again using the EXP operation. These lists are identified by a label of the form $nnnnn were nnnnn is the index of the statement table associated with the label list. There is no need for a special list terminator, the statement table entry contains a count or vector length for its label list, if any.

12. Appendix V - The SIMISP.REQ File

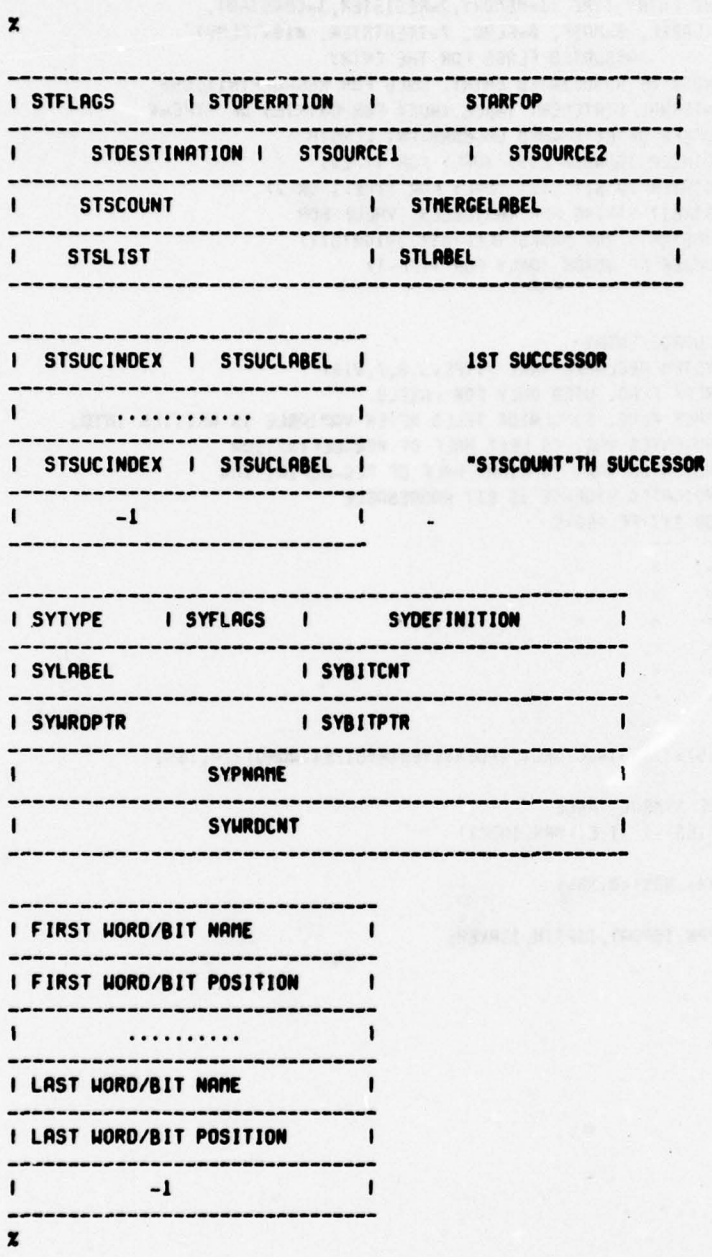
12.1. The Statement Table

```
MACRO
        STFLAGS=0,27,9$,          !ASSORTED FLAGS FOR THE STATEMENT
        STOPERATION=0,18,9$,      !OPERATION CODE. SEE XTOP.REQ
        STARFOP=0,0,18$,          ! ARF OPERATION CODE
        STDESTINATION=1,24,12$,   !DESTINATION VARIABLE SYMBOL TABLE INDEX
        STSOURCE1=1,12,12$,       !SOURCE1 VARIABLE SYMBOL TABLE INDEX
        STSOURCE2=1,0,12$,        !SOURCE2     "        "     "     "
        STSCOUNT=2,18,18$,        !NUMBER OF ELEMENTS IN STSLIST.
        STLABEL=3,0,18$,          !SYMBOL TABLE INDEX OR 0.
        STMERGELABEL=2,0,18$,     !LABEL OF THE ASSOC. MERGE STATEMENT FOR
                                  !XTDIVERGE ,XTBRANCH  AND XTCALL OPS.
                                  !LABEL OF ASSOC. STATEMENT FOR XTCALLOP.
        STSLIST=3,18,18$;         !POINTER TO VECTOR OF SUCCESSOR STATEMENTS.
                                  !STSUCSTRUCT IS MAPPED ONTO THE VECTOR

BIND    !THE STTABLE FLAGS
        STUSERLAB=1^0,            !STATEMENT LABEL WAS DECLARED BY USER
        STBREAK=1^1,              !BREAK FLAG. SIMULATOR BREAKS AFTER FLAGGED
                                  !STATEMENTS ARE EXECUTED
        STTRACE=1^2,              !TRACE FLAG. SIMULATOR WILL PRINT VARIABLES
                                  ! AFTER EXECUTION.
        STRECORD=1^3,             !RECORD THE SIMULATED TIME OF EACH EXECUTION
        STIGNORED=1^4,            !FLAGS DIVERGE,PMERGE AND ASSOC. JOINS AS
                                  !DELETED STATEMENTS!!
        STOPAQUE=1^5,             ! DISABLES READ/WRITE/ACCESS TALLY
        STETCETC=0;               !ADD ANY OTHER FLAGS YOU LIKE

BIND
        STENTRYSIZE=4;            !4 WORDS/ENTRY

STRUCTURE STSTRUCTURE(INDEX,WORD,P,S)=(.STSTRUCTURE+.INDEX*STENTRYSIZE+.WORD)<.P,.S>;

EXTERNAL STSTRUCTURE STTABLE;   !THE STATEMENT TABLE
EXTERNAL STTOP;         !THE INDEX OF THE LAST STTABLE ENTRY (STARTING FROM 0)


MACRO
        STSUCLABEL=18,18$,        !THE SUCCESSOR LABEL
        STSUCINDEX=0,18$;         !THE SUCCESSOR INDEX

STRUCTURE STSUCSTRUCT(WORD,P,S)=(.STSUCSTRUCT+.WORD)<.P,.S>;
```

12.2. The Symbol Table

```
MACRO
        SYTYPE=0,27,9$,         !THE ENTRY TYPE (1=MEMORY,2=REGISTER,3=CONSTANT,
                                !4=LABEL, 5=MASK, 6=FLAG, 7=TREGISTER, #10=TFLAG)
        SYFLAGS=0,18,9$,                !ASSORTED FLAGS FOR THE ENTRY
        SYDEFINITION=0,0,18$,   !INDEX OF ASSOCIATED ENTRY. USED FOR REG-DEFINITIONS
        SYLABEL=1,18,18$,       !INTERNAL STATEMENT TABLE INDEX FOR ENTRIES OF TYPE=4
        SYBITCNT=1,0,18$,       !NUMBER OF BITS/WORD OR CONSTANT LENGTH
        SYWRDPTR=2,18,18$,      !POINTER TO WORD LIST (ONLY FOR TYPE=1)
        SYBITPTR=2,0,18$,       !POINTER TO BIT LIST (ONLY FOR TYPE=1 OR 2)
        SYPNAME=3,0,36$,        ! SIXBIT STRING FOR VARIABLES, VALUE FOR
                                !CONSTANTS AND MASKS (LEFTBIT,,RIGHTBIT)
        SYWRDCNT=4,0,36$;       !NUMBER OF WORDS (ONLY FOR TYPE=1)

BIND
        SYENTRYSIZE=5,          !5 WORDS/ENTRY
        SYSYSTEMVAR=1^0,        !SYSTEM DECLARED VAR. (TYPE=3,5,7,#10)
        SYBREAK=1^1,            !BREAK FLAG. USED ONLY FOR LABELS.
        SYTRACE=1^2,            !TRACE FLAG. SIMULATOR TELLS AFTER VARIABLE IS WRITTEN INTO.
        SYPRIMARY=1^4,          !INDICATES VAR. IS LEFT HALF OF REG-DEFINITION
        SYSECONDARY=1^5,        !INDICATES VAR. IS RIGHT HALF OF REG-DEFINITION
        SYBITADDRESS=1^6,       !INDICATES STORAGE IS BIT ADDRESABLE
        TYPEMEMORY=1,           !FOR SYTYPE ABOVE
        TYPEREGISTER=2,         ! "     "       "
        TYPECONSTANT=3,         ! "     "       "
        TYPELABEL=4,            ! "     "       "
        TYPEMASK=5,             ! "     "       "
        TYPEFLAG=6,             ! "     "       "
        TYPETREGISTER=7,        ! "     "       "
        TYPETFLAG=8;            ! "     "       "

STRUCTURE SYSTRUCTURE(INDEX,WORD,P,S)=(.SYSTRUCTURE+.INDEX*SYENTRYSIZE+.WORD)<.P,.S>;

EXTERNAL SYSTRUCTURE SYTABLE;   !THE SYMBOL TABLE
EXTERNAL SYTOP; !THE NUMBER OF ENTRIES -1 (I.E. MAX INDEX)

STRUCTURE IVECTOR(NDX)=(I)(..IVECTOR+.NDX)<0,36>;

EXTERNAL ISPTIT,ISPFNAM,ISPEXT,ISPPPN,ISPDAT,ISPTIM,ISPVER;
```

12.3. Table Diagrams



A User's Guide to the ISPL Simulator

Mario R. Barbacci
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa.

TABLE OF CONTENTS

Abstract

The simulator described in this manual will interpret the output of the ISPL compiler, the RTM code, thus allowing the users a generalized computer architecture simulation facility. This manual describes the commands available to the users.

Acknowledgements

The ISP simulator is a much improved version of a primitive system developed by S. Rodkey at CMU during the spring of 1975. The system was modified and expanded by Greg Lloyd of the Naval Research Laboratory during the Fall of 1975. The system was further enhanced by the author during the Winter and Spring of 1976. Many commands and features were added to the system as part of the Army/Navy CFA project. Special thanks are due to the users of the system for their comments and suggestions, among them: H. Elovitz (NRL), R. Gordon (NUSC), R. Howbrigg (NUSC), D. Siewiorek (CMU), and S. Zuckerman (NRL).

1. Introduction

The ISPL compiler translates Computer Architecture (Instruction Set) Descriptions written in a subset of ISP [Bell71] into instructions for an idealized Register Transfer Machine (RTM) which can perform the primitive Register Transfer Operations needed to fetch, decode, and execute instructions. The ISP simulator is in effect an implementation of the Register Transfer Machine.

Some effort has been put into isolating the user from the low level detail of the RTM code. Under normal circumstances, the user will interact with the simulator using the names of registers, memories, procedures, etc, as declared in the ISPL description.

The simulator follows the convention of the ISPL compiler with regard to number representation, it uses an unsigned (pure magnitude) representation. Internally, the simulator uses multiple precision operations on the PDP-10 to execute the data operations and transfers. A current implementation limitation sets a limit of 140 bits for the length of the variables used in the register transfer operations (beware that the ISPL compiler will allow the user to declare registers and memories of arbitrary length - the simulator will warn the user if any attempt is made to operate on variables larger than 140 bits).

Although concurrency is easily described in ISPL, the simulator makes no attempt to provide this facility. It will execute concurrent operations in sequence and the user should avoid writting order-dependent parallel ISP statements.

2. From ISPL to RTM and Beyond

The process of obtaining a running simulator given a syntactically correct ISP description is rather simple. The ISPL compiler, in the abscence of serious errors, will produce a MACRO10 program containing the RTM object code. This program should be assembled in order to produce a relocatable PDP-10 binary file. This process is also handled by the ISPL compiler (i.e. it will generate the RTM file and then invoke the MACRO10 assembler). At the end of the compilation the user has the following files (assume that the original ISP files was X.ISP):

X.ISP (The source file)
X.LST (The listing file, described in the ISPL compiler manual)
X.RTM (The object file, described in the ISPL compiler manual)
X.REL (The relocatable binary version of the X.RTM file)

At this point you can get rid of the X.LST and X.RTM files, as far as the simulator is concerned, they are not needed at all. Hold onto the X.REL file for dear life, unless cycles are cheap at your installation and can afford to run the ISPL compiler as often as you please....

The simulator consists of a group of (currently 7) binary files that must be linked, using one of the standard PDP-10 CUSPs, with the X.REL file. Once this is done, you can save the core image and you are all set to go. The exact procedure might change from installation to installation, depending on whether you use LOAD or LINK10.

A typical procedure might look like:

```
EXECUTE X.REL,@ISPSIM.CMD

<or alternative, if you have the LINK-10 loader in your system:>

R LINK
*x.REL
*@ISPSIM.CMD
*<any switches you want>
*/SSAVE x
```

```
*/GO
```

The above sequence will produce two files: X.SHR and X.LOW. These are your ISPL description compiled, linked, saved, and ready to run:

```
RU x < and off you go!, good luck!>
```

3. The Command Language

The simulator accepts a small number of commands, using a fixed format:

<keyword> <parameter> <parameter> ....

Only one command is accepted per line. Commands might be typed, in upper or lower case, directly from the user's terminal or can be retrieved from command files (the latter can be done recursively, up to 16 levels of nested command files). Comments can be inserted in the command stream by typing a "!" followed by any arbitrary string. The command scaner will ignore anything between the "!" and the end of the line. Most parameters represent ISPL variable names or numeric values. The latter can be typed in several modes (Binary, Octal, Decimal, and Hexadecimal) and there are facilities to set up a proper default value of the type-in type-out radix.

All variables, labels, and constants defined in the ISP source program have activity counters associated with them. This allows the user to collect statistical data when running benchmark programs under the simulator. There are commands to clear, preset, and interrogate the value of these counters.

The command language include a group of commands to trace variables, start, break, and continue a simulation run, as well as commands to set and interrogate the values of the register and memories of the target machine.

When the simulator is running and the user suspects an infinite loop of instructions, typing a $ (Altmode) will break the execution. Actually, any type ahead will produce an interruption. $ is the preferred mode.

3.1. START and CONTinue

START <label> is the command used to begin the simulation of an ISP procedure or main program. <label> is the name of a procedure declared in the ISP description. The START command is valid only at the top level of simulation. Thus, after a breakpoint in the simulation the user must use the command CONT to proceed.

3.2. EXIT

EXIT is the command used to finish a simulation run. It allows an orderly return to the PDP-10 monitor. EXIT closes the files that might have been created with the OCONNECT command. Typing ↑C will return to the monitor but CONNECTed files will be lost.

3.3. READ and DUMP

READ <dev:filename.ext[ppn]> allows the user to specify a file containing simulation commands. Essentially, READ substitutes the user terminal with the file and proceeds to read and execute commands until the end of the file is found, at which point the user terminal is again the command input device. Defaults are DSK (device), SIM (extension) and current user's PPN. Command files can contain comments. A comment is anything between a ! and the end of a line.

DUMP is used to save the status of a simulation run. DUMP creates a file containing the values of each variable (if non-zero), trace/break flags, read/write counters, etc. The file created by DUMP can be read by the READ command, thus allowing a simple way of reinitializing a simulation at the point the DUMP command was issued.

3.4. ECHO and DECHO

ECHO and DECHO are commands used to set an internal flag that controls the ECHOing of the commands being read from a command file onto the user terminal. After the ECHO command is issued, the execution of a READ command will type onto the user"s terminal the command lines as they appear in the command file. DECHO disables this type-out. ECHO and DECHO can be issued from inside the command file thus allowing a selective type-out.

3.5. RADIX

RADIX <base> is used to set the numeric base to be used for typing in and out. <base> is one of the following strings: BINARY, OCTAL, DECIMAL, or HEX. If base is ommited the command simply types the name of the current base without altering it. The current base setting might be bypassed on input by prefixing the constant with one of the following: ' (binary), @ (octal) or " (hex). Regardless of the current radix, HEX constants which begin with a letter MUST be prefixed with " (this is a requirement that will be lifted in a future release).

3.6. CTR, SETCTR, and OUTCTR

CTR <name> displays the value of the counter(s) associated with <name>. These counters are tagged with R, W, or L to indicate whether they are the Read, Write, or Label count respectively. SETCTR <name> <readcounter> <writecounter> allows the user to specify the setting of these counters. If <name> is a label, then the <readcounter> plays the role of label count. If the <...counter> values are ommitted they default to 0. Instead of <name> the user may specify ALL and the command is

applied to all the variables and labels. All read/write counts are expressed in terms of 8-bit bytes. Thus, reading a 16 bit register increments the R counter by 2. The register lengths are rounded up to the next multiple of 8 before incrementing the counter: A 19 bit register counts as 3.

OUTCTR <filename.ext[ppn]> is a subset of the DUMP command. It creates a file (default extension CTR) with the values of of all non-zero counters.

3.7. OPAQUE and DOPAQUe

OPAQUE <label-list> and DOPAQUE <label-list> are used to inhibit or enable the variable and label activity counters. The parameters to these two commands are labels or procedure names. If a procedure is OPAQUEd then no activity counts are incremented during its execution. The DOPAQUE command re-enables the activity counting. These two commands affect only those procedures named in the parameter list. Procedures called by OPAQUEd or DOPAQUEd procedures are not affected.

3.8. VALUE and SETVALue

VALUE and SETVALue are the commands used to set and interrogate the contents of the ISP variables. The valid formats are:

VALUE <regname> (displays the value of a single register)

VALUE <memname> [<fromword> {: <toword>}] (displays the values stored in a memory).

SETVAL <regname> = <value> (stores <value> into the register)

SETVAL <memname> [<fromword>] = <value-list> (stores into the memory. If more than one value is specified, they are stored in succesive memory positions, starting at <fromword>).

3.9. TRACE, UTRACE, DTRACE, and TELLTRace

{TRACE | UTRACE | DTRACE} <variable-list> are the commands used to enable or disable the tracing of variables during the simulation. If the identifier ALL is specified instead of a variable list, the command applies to all variables. TRACE and UTRACE differ in that the former applies to all variables (including compiler declared temporary registers and flags) while the latter only applies to user declared variables (registers and memories). DTRACE is used to disable the tracing.

TELLTRace will type on the user's terminal the list of variables currently being traced.

3.10. BREAK, DBREAK, and TELLBReak

{BREAK | DBREAK} <label-list> are the commands used to enable or disable the setting of breakpoints during the simulation. The parameters are either ISP procedure names or labels. TELLBR displays on the user's terminal the list of breakpoint names.

3.11. SBREAK, DSBREAK, and TELLSBreak

These commands are similar to BREAK, DBREAK, and TELLBReak but instead of using ISP labels as parameters they take RTM statement numbers. Thus allowing a finer degree of control on the placement of the breakpoints. These commands are not particularly useful for the normal user, who should not be concerned with the RTM code.

3.12. ICONNEct and OCONNEct

ICONNEct <identifier>,<channel-number>,<variable-name>

OCONNEct <identifier>,<channel-number>,<variable-name>

These commands are used to "connect" ISP variables to PDP-10 ASCII files which will act as potentially infinite sources/sinks for variable values. When a variable is connected to an input file, each time the variable is accessed, the value will be obtained from the file instead of the simulated storage allocated to the variable. Similarly, writting into a variable that has been connected to an output file results in the value being written into the file (as well as into the storage allocated to the variable). The format for both input and output files is the same: one number/line.

The file names are created by the simulator and consist of the first parameter to the command (the <identifier>) as the file name, with extension ICn (ICONNEct) or OCn (OCONNEct), where n is the user specified channel number. The current implementation only allows up to three input and three output channels open simultaneoulsy. Thus the only valid channel numbers are 1, 2 and 3.

3.13. HELP

HELP tells the user about the command names and their format. HELP <commandname> tells the user about a specific command.

4. Storage Mapping

The simulator allocates space for the registers and memories declared in the RTM symbol table using contiguous storage on the memory of the PDP-10. The fact that the PDP-10 is a 36 bits/word, 2's complement machine is completely transparent to the user. All RTM operations are interpreted rather than compiled into PDP-10 instructions. Moreover, the simulator does not impose any limitations derived from the word length; ISPL registers and memories are allocated contiguous bit strings on the PDP-10.

The use of logical register/memory declarations in the ISPL description presents the following problem: The ISPL compiler allows the user to define arbitrary mappings between bits of the left and right hand sides of the logical declaration, the only check made at that point is that the number of bits is the same. From the simulator point of view, it could be posible to implement arbitrary bit mappings at a tremendous degradation in performance (accessing a bit of a register or memory word that is mapped onto some other component implies searching a table of bit name/position equivalences; having to follow this procedure bit by bit, even for full register/word accesses could be hard to justify). The simulator makes a compromise between convenience to the ISPL writer and efficiency of simulation. The solution adopted is to restrict the types of mappings that the simulator can handle: all the bits of the right hand side of a logical declaration must be contiguous. Continuity is defined in terms of the word/bit naming convention used in the main declaration of the register/memory used on the right hand side of the logical declaration. There are no limitations as to what can appear on the left hand side of the logical declaration, these bits are by definition contiguous.

Specifically, the following are the valid types of mappings allowed by the simulator:

1) If the right hand side of a mapping was declared as a register, the structure of the right hand side must specify a contiguous string of bit names as specified in the main declaration. The number of bits may range from 1 to the entire register length and, for proper subsets of the main declaration, may be located anywhere in the register.

2) If the right hand side of a mapping consists of a single memory word, the valid mappings are those defined as above.

3) If the right hand side of a mapping consists of a set of memory words, the structure of the right hand side must specify a contiguous string of full words as specified in the main declaration. The number of words may range from 1 to the entire memory range and, for proper subsets of the main declaration, may be located anywhere in the memory.

4.1. Allowable Types of Mapping

The following list of memory maps gives a good coverage of the allowable cases:

```
M[#777777:#770000,#7777:0]<7:0>;          !THE ADDRESSING SPACE
MB[#7777:0]<7:0>                :=        M[#7777:0]<7:0>;
MBIO[#777777:#770000]<7:0>      :=        M[#777777:#770000]<7:0>;
MW[#3777:0]<15:0>               :=        M[#7777:0]<7:0>;
MWIO[#377777:#370000]<15:0>     :=        M[#777777:#770000]<7:0>;

A0N0N[0:255]<0:15>;
A0NN0[0:255]<15:0>;
AN00N[255:0]<0:15>;
AN0N0[255:0]<15:0>;
R0N<0:15>;
RN0<15:0>;

MAP11[0:15]<0:15>       :=      A0N0N[100:115]<0:15>;
MAP12[0:15]<0:15>       :=      A0NN0[100:115]<15:0>;
MAP13[0:15]<0:15>       :=      AN00N[115:100]<0:15>;
MAP14[0:15]<0:15>       :=      AN0N0[115:100]<15:0>;
MAP15[0:2]<0:2>         :=      R0N<5:13>;
MAP16[0:2]<0:2>         :=      RN0<13:5>;

MAP21[0:15]<15:0>       :=      A0N0N[100:115]<0:15>;
MAP22[0:15]<15:0>       :=      A0NN0[100:115]<15:0>;
MAP23[0:15]<15:0>       :=      AN00N[115:100]<0:15>;
MAP24[0:15]<15:0>       :=      AN0N0[115:100]<15:0>;
MAP25[0:2]<2:0>         :=      R0N<5:13>;
MAP26[0:2]<2:0>         :=      RN0<13:5>;

MAP31[15:0]<0:15>       :=      A0N0N[100:115]<0:15>;
MAP32[15:0]<0:15>       :=      A0NN0[100:115]<15:0>;
MAP33[15:0]<0:15>       :=      AN00N[115:100]<0:15>;
MAP34[15:0]<0:15>       :=      AN0N0[115:100]<15:0>;
MAP35[2:0]<0:2>         :=      R0N<5:13>;
MAP36[2:0]<0:2>         :=      RN0<13:5>;

MAP41[15:0]<15:0>       :=      A0N0N[100:115]<0:15>;
MAP42[15:0]<15:0>       :=      A0NN0[100:115]<15:0>;
MAP43[15:0]<15:0>       :=      AN00N[115:100]<0:15>;
MAP44[15:0]<15:0>       :=      AN0N0[115:100]<15:0>;
MAP45[2:0]<2:0>         :=      R0N<5:13>;
MAP46[2:0]<2:0>         :=      RN0<13:5>;

MAP51<0:5>      :=      A0N0N[100]<4:9>;
MAP52<0:5>      :=      A0NN0[100]<9:4>;
MAP53<5:0>      :=      R0N<5:10>;
MAP54<5:0>      :=      RN0<10:5>;
```

5. Examples

This section contains the transcript of several actual runs. The first example is based on the small ISPL example described in the ISPL manual. The transcript for the compilation phase of the multiplier example appears in the ISPL compiler manual. We start from the point right after the MACRO10 assembler has generated the *.REL file.

5.1. Linking the Compiler Output with the Simulator

```
r link
*mult
*@ispsim
*/ssave mult
*/go

EXIT
```

MULT.REL is the name of the file created by the ISPL compiler. ISPSIM.CMD is the name of the command file containing the list of files that make up the simulator. It also contains commands to load the BLISS10 run time library. The use of the SSAVE switch instead of the SAVE switch creates a shareable version of the program. Thus the result of the LINK10 execution will be named MULT.SHR+MULT.LOW.

5.2. Running the Simulator

Here we run the program that was created in the previous transcript. The example makes use of a few simple commands that set initial values in the variables, selects some variables for tracing and then starts the execution at the main entry point of the description. The example is simple and self explanatory.

```
ru mult
ISP SIMULATOR V3 - NRL ARF STAGE 2
Thursday 29 Jul 76 23:42:13 MULT.ISP[N655M025]
SERIALIZATION COMPLETED
SPACE ALLOCATED
TYPE HELP FOR HELP
TYPE <ESC> TO INTERRUPT SIMULATION LOOPS

>radix octal
>setval p←2
>setval mpd←3000                    ! #6 on left half of mpd
>trace mpd,p,c
>start 10

@ L0      ←#2     C       =#10
@ STEP    ←#4     P       =#1
@ L1      ←#4     C       =#7
@ STEP    ←#10    P       =#1400
@ L1      ←#4     C       =#6
@ STEP    ←#4     P       =#600
@ L1      ←#4     C       =#5
@ STEP    ←#4     P       =#300
@ L1      ←#4     C       =#4
@ STEP    ←#4     P       =#140
@ L1      ←#4     C       =#3
@ STEP    ←#4     P       =#60
@ L1      ←#4     C       =#2
@ STEP    ←#4     P       =#30
@ L1      ←#4     C       =#1
@ STEP    ←#4     P       =#14
@ L1      ←#4     C       =#0
SIMULATION COMPLETED

RUN TIME(10 usec units)=45259
RTM OPS EXECUTED=136

>value p
P       =#14
>value mpd
MPD     =#3000
>exit
EXIT
```

When the simulator starts it performs two preliminary operations: 1) It transforms the RTM statement table eliminating the DIVERGE/PMERGE operations that define concurrent operations, and 2) It allocates space for the registers and memories declared in the RTM symbol table. The simulator then types two messages advising the user of the existance of the HELP command and of the use of the <ESC> (AltMode) to break the execution of the simulator from the user's terminal.

The tracing of variables indicates the place in the ISPL program where an assignment to the variable has occurred. The location is identified by printing the nearest ISPL label together with a displacement (in RTM operations) from this label. The name of the variable affected by the transfer is printed, together with the new value. The run time printed at the end of the simulation is obtained from a fast 10us. clock available at CMU. Some installations might now have this feature.

In the above example we initialize the multiplier (P) to 2 and the multiplicand (MPD) to 6. According to the algorithm, the multiplicand is stored in the left half of the MPD register. In the current implementation of the simulator we can not specify partial register initialization, thus, we have to load the right half of MPD with a suitable value (initialization of variables in the command language implies full register modification, with zeroes on the left of the value). At the end of the run, the contents of the P register contains the result of the multiplication (6*2=12 or *14 given that we set the type out radix to OCTAL).

5.3. Executing Selected Procedures

In the following example we show a few more commands and features of the simulator:

```
ru mult
ISP SIMULATOR V3 - NRL ARF STAGE 2
Thursday 29 Jul 76 23:42:13 MULT.ISP[N655MB25]
SERIALIZATION COMPLETED
SPACE ALLOCATED
TYPE HELP FOR HELP
TYPE <ESC> TO INTERRUPT SIMULATION LOOPS

>radix octal

>setval p=3

>setval mpd=400          ! Multiplicand=1

>utrace all

>start step

@ STEP   +#10   P      =#201
RUN TIME(10 usec units)=3001
RTM OPS EXECUTED=9
```

The above sequence shows how the simulator can be used to execute selected procedures from the ISPL description. In fact, the simulator treats ALL labels and procedure names as potential entry points. It does not assign any special meaning to the label of the main body of the ISPL description.

5.4. Reading Command Files

The following example shows the use of the READ command. In this particular case we are not only initializing the variables and setting trace flags, but we are also starting the simulation automaticaly from the command file. The number of ">" character used to prompt the input stream (a user or a command file) indicates the level of nesting of the command stream. One ">" is the mark of the top level.

```
>dtrace all

>read m1.sim

>>! this is a command file

>>setval p=2

>>setval mpd=2000
>>       ! multiplicand=4

>>trace p

>>start 10

@ STEP    +#4     P       =#1
@ STEP    +#10    P       =#1000
@ STEP    +#4     P       =#400
@ STEP    +#4     P       =#200
@ STEP    +#4     P       =#100
@ STEP    +#4     P       =#40
@ STEP    +#4     P       =#20
@ STEP    +#4     P       =#10
SIMULATION COMPLETED

RUN TIME(10 usec units)=32120
RTM OPS EXECUTED=136

>>!end of command file

>>7 LINES READ

>exit

EXIT
```

S360 ISP DESCRIPTION of the IBM S/360, Interdata 8/32, and DEC PDP-11

```
! ISP DESCRIPTION OF IBM SYSTEM/360 ARCHITECTURE
!
! THIS DESCRIPTION INCLUDES THE STANDARD INSTRUCTION SET ONLY.
! THE FLOATING-POINT FEATURE INSTRUCTIONS AND DECIMAL FEATURE
!  INSTRUCTIONS ARE NOT DESCRIBED.
! THE PROTECTION FEATURE INSTRUCTIONS AND DIRECT CONTROL FEATURE
!  INSTRUCTIONS ARE DESCRIBED.
! THE TEST AND SET INSTRUCTION IS NOT DESCRIBED DUE TO THE
!  LIMITATIONS OF A SINGLE ISP DESCRIPTION TO COVER TWO INDEPENDENT
!  PROCESSES.  A SECOND PROCESS (ISP DESCRIPTION) SHOULD BE GIVEN
!  FOR THE MEMORY LATCHING.
! THE DIAGNOSE INSTRUCTION DESCRIPTION ( WHICH IS A MODEL DEPENDENT
!  INSTRUCTION ) WAS MODIFIED FOR USE AS A HALTING MECHANISM FOR
!  THE SIMULATION.  THEREFORE, THE DIAGNOSE INSTRUCTION DOES NOT
!  CORRESPOND TO ANY S/360 MODEL DIAGNOSE INSTRUCTION.
! THE CLCL ( COMPARE LOGICAL LONG ) INSTRUCTION FROM THE S/370
!  WAS ADDED FOR RUNNING BENCHMARKS.  IT IS NOT A TRUE DESCRIPTION
!  OF THE INSTRUCTION SINCE IT IS NOT INTERRUPTABLE.
! ADDITIONAL LABELS WERE ADDED TO AID IN MEASURING THE BENCHMARK
!  PROGRAMS.  THESE ARE NOT PART OF THE ARCHITECTURE DESCRIPTION.
!

S360:=
        (DECLARE

        MACRO MAXDW:=2047 $
        MACRO MAXW:=4095 $
        MACRO MAXH:=8191 $
        MACRO MAXB:=16383 $
        MACRO MAXKEY:=7 $
        MACRO BEGIN:=( $
        MACRO END:=) $

! PRIMARY MEMORY

        MEMDW[0:MAXDW]<0:63>;                          !DOUBLEWORD MEMORY
        MEMW[0:MAXW]<0:31>:=MEMDW[0:MAXDW]<0:63>;      !WORD MEMORY
        MEMH[0:MAXH]<0:15>:=MEMW[0:MAXW]<0:31>;        !1/2 WORD MEMORY
        MEMB[0:MAXB]<0:7>:=MEMH[0:MAXH]<0:15>;         !BYTE MEMORY

        STKEYS[0:MAXKEY]<0:4>;                         ! STORAGE KEY ARRAY

! PERMANENT STORAGE ASSIGNMENTS

        IPLPSW<0:63>:=MEMB[0:7]<0:7>;          ! IPL PSW
        IPLCW1<0:63>:=MEMB[8:15]<0:7>;         ! IPL CCW #1
        IPLCW2<0:63>:=MEMB[16:23]<0:7>;        ! IPL CCW #2
        EXOPSW<0:63>:=MEMB[24:31]<0:7>;        ! EXTERNAL OLD PSW
        SVCPSW<0:63>:=MEMB[32:39]<0:7>;        ! SVC OLD PSW
        PROPSW<0:63>:=MEMB[40:47]<0:7>;        ! PROGRAM OLD PSW
        MKOPSW<0:63>:=MEMB[48:55]<0:7>;        ! MACHINE CHECK OLD PSW
        IOOPSW<0:63>:=MEMB[56:63]<0:7>;        ! I/O OLD PSW
        CHSTWD<0:63>:=MEMB[64:71]<0:7>;        ! CHANNEL STATUS WORD
        CHADWD<0:31>:=MEMB[72:75]<0:7>;        ! CHANNEL ADDRESS WORD
        TIMER<0:23>:=MEMB[80:82]<0:7>;         ! TIMER CELL
        EXNPSW<0:63>:=MEMB[88:95]<0:7>;        ! EXTERNAL NEW PSW
        SVNPSW<0:63>:=MEMB[96:103]<0:7>;       ! SVC NEW PSW
        PRNPSW<0:63>:=MEMB[104:111]<0:7>;      ! PROGRAM NEW PSW
        MKNPSW<0:63>:=MEMB[112:119]<0:7>;      ! MACHINE CHECK NEW PSW
        IONPSW<0:63>:=MEMB[120:127]<0:7>;      ! I/O NEW PSW
        SCNOUT<0:63>:=MEMB[128:135]<0:7>;      ! DIAGNOSTIC SCAN OUT

!PROCESSOR STATE

        REG[0:15]<0:31>;          ! GENERAL PURPOSE REGISTERS
        PSW<0:63>;                ! PROGRAM STATUS WORD
        PSWW[0:3]<0:15>:=PSW<0:63>;      ! ALTERNATE PSW DEFINITION
                CHAMSK<0:7>:=PSW<0:7>;   ! CHANNEL MASK
                PROTKY<0:3>:=PSW<8:11>;  ! PROTECTION KEY
                ASCMSK<>:=PSW<12>;       ! USASCII MASK
                MCHKMK<>:=PSW<13>;       ! MACHINE CHECK MASK
                WAITST<>:=PSW<14>;       ! WAIT STATE
                PROBST<>:=PSW<15>;       ! PROBLEM STATE
                INTCDE<0:15>:=PSW<16:31>;       !INTERRUPT CODE
```

```
ILC<0:1>:=PSW<32:33>;     ! INSTRUCTION LENGTH CODE
CC<0:1>:=PSW<34:35>;      ! CONDITION CODE
FPOPMS<>:=PSW<36>;        ! FIXED POINT OVERFLOW MASK
DOFMSK<>:=PSW<37>;        ! DECIMAL OVERFLOW MASK
EXOFMS<>:=PSW<38>;        ! EXPONENT UNDERFLOW MASK
SIGMSK<>:=PSW<39>;        ! SIGNIFICANCE MASK
PC<0:23>:=PSW<40:63>;     ! PROGRAM COUNTER (24 BITS)
```

AD-A049 483 ARMY ELECTRONICS COMMAND FORT MONMOUTH N J F/G 9/2
COMPUTER FAMILY ARCHITECTURE SELECTION COMMITTEE FINAL REPORT. --ETC(U)
SEP 77 M BARBACCI, R GORDON, R HOWBRIGG

UNCLASSIFIED ECOM-4529 NL

2 OF 3
AD
A049483

```
S360 ISP DESCRIPTION

!IMPLEMENTATION RELATED VARIABLES
!
! THESE DECLARATIONS AND DEFINITIONS ARE NOT ACTUALLY PART OF THE
!  ARCHITECTURE DESCRIPTION, BUT ARE NECESSARY FOR THE ISP
!  DESCRIPTION.
!

        IR<0:47>;                       ! INSTRUCTION REGISTER
        IRW[0:2]<0:15>:=IR<0:47>;       ! 1/2 WORD ADDRESS FOR IR (IN EXECUTE)
                OPCODE<0:7>:=IR<0:7>;   ! RR,RX,RS,SI,SS
                R1<0:3>:=IR<8:11>;      ! RR,RX,RS
                R2<0:3>:=IR<12:15>;     ! RR
                X2<0:3>:=IR<12:15>;     ! RX
                B1<0:3>:=IR<16:19>;     ! RX,RS,SI,SS
                D1<0:11>:=IR<20:31>;    ! RX,RS,SI,SS
                R3<0:3>:=IR<12:15>;     ! RS
                M1[0:3]<>:=IR<8:11>;    ! MASK 1
                I2<0:7>:=IR<8:15>;      ! SI
                LFLD<0:7>:=IR<8:15>;    ! SS
                L1<0:3>:=IR<8:11>;      ! SS
                L2<0:3>:=IR<12:15>;     ! SS
                B2<0:3>:=IR<32:35>;     ! SS
                D2<0:11>:=IR<36:47>;    ! SS
        MAR<0:23>;                      ! MEMORY ADDRESS REGISTER
        AMAR1<0:23>;                    ! AUXILLARY MEMORY ADDRESS REG.(1)
        AMAR2<0:23>;                    ! AUXILLARY MEMORY ADDRESS REG.(2)
        MBR<0:31>;                      ! MEMORY BUFFER REGISTER
        LOBYTE<0:7>:=MBR<0:7>;          ! LEFT BYTE IN MBR
        HIBYTE<0:7>:=MBR<24:31>;        ! RIGHT BYTE IN MBR
        LAUX1<0:7>;                     ! BYTE COUNT REGISTER 1
        LAUX2<0:7>;                     ! BYTE COUNT REGISTER 2
        TEMP<0:16>;                     ! 17 BIT TEMP
        DIVREG<0:63>;                   ! 64 BIT DIVIDEND REGISTER
        EXRF<>;                         ! EXECUTE RECURSION FLAG
        ZONE<0:3>;                      ! ZONE TEMPORARY
        DIGIT<0:3>;                     ! DIGIT TEMPORARY
        SCALE<0:63>;                    ! SCALE FACTOR FOR CVB
        T0<>;                           ! NO-OP REGISTER
        T1<>;                           ! 1 BIT TEMP
        T1A<>;                          ! 1 BIT AUXILIARY TEMP
        T2<0:1>;                        ! 2 BIT TEMP
        T2A<0:1>;                       ! 2 BIT AUXILLIARY TEMP
        T4<0:3>;                        ! 4 BIT TEMP
        T6<0:5>;                        ! 6  "   "
        T8<0:7>;                        ! 8  "   "
        T8A<0:7>;                       ! 8  "  AUXILLIARY TEMP
        T16<0:15>;                      ! 16 "   TEMP
        T24<0:23>;                      ! 24 "   "
        T32<0:31>;                      ! 32 "   "
        T33<0:32>;                      ! 33 "   "
        T64<0:63>;                      ! 64 "   "
        OVF<>;                          ! OVERFLOW
        STOPBIT<>;                      ! STOP SWITCH
        INTVEC<0:4>;                    ! INTERRUPT VECTOR
                                        ! BIT 0 = MACHINE CHECK
                                        ! BIT 1 = SVC
                                        ! BIT 2 = PROG CHECK
                                        ! BIT 3 = EXTERNAL INTERRUPT (TIMER)
                                        ! BIT 4 = I/O INTERRUPT
        IOMSK<0:7>;                     ! CHANNEL MASK REGISTER
        CHRLS<>;                        ! CHANNEL RELEASE
        CHSEL<>;                        ! CHANNEL SELECT REGISTER
        CHANCC<0:1>;                    ! CHANNEL CONDITION CODE
        CHINST[0:3]<>;                  ! CHANNEL INSTRUCTION LINE
                                        ! 0 => SIO
                                        ! 1 => TIO
                                        ! 2 => HIO
                                        ! 3 => TCH
        CHAREG<0:7>;                    ! CHANNEL ADDRESS REGISTER
        DEVREG<0:7>;                    ! DEVICE REGISTER
                                        ! HOLDS DEVICE ADDRESS (0-255)
        EXTREG<0:7>;                    ! EXTERNAL REGISTER
                                        ! BIT 0 = TIMER INTERRUPT
                                        ! BIT 1 = CONSOLE INTERRUPT
```

```
S360 ISP DESCRIPTION

        IODREG<0:7>;                    ! HOLDS DATA BYTE FOR DIRECT I/O
                                        ! ITS MEANING (COMMAND OR DATA)
                                        ! IS IMPLEMENTATION DEPENDENT
        SIGOUT<0:9>;                    ! SIGNAL OUT FOR DIRECT I/O

        MACRO NOP:=T0←0 $
```

```
S360 ISP DESCRIPTION

! UTILITY ROUTINES

! PRIVILEGED STATE CHECK ROUTINE

        PSCHK:=
                BEGIN
                IF PROBST => INTCDE←2; INTVEC<2>←1 NEXT BAILOUT ICYCLE
                END;

! INTERRUPT CODE 5 IMPLIES ADDRESSING ERROR
! INTERRUPT CODE 6 IMPLIES SPECIFICATION (ALIGNMENT ERROR)
! INTERRUPT CODE 4 IMPLIES PROTECTION
! THE ORDER OF SETTING THESE CODES MAY BE IMPLEMENTATION DEPENDENT
! TESTS ON A MODEL 75 SHOW CODE 6 IS FIRST

! CHECK ROUTINE FOR STORAGE PROTECTION

        CKPR:=
                BEGIN
                IF STKEYS[MAR<0:12>]<1:4> NEQ PROTKY =>
                        INTVEC<2>←1; INTCDE←4 NEXT BAILOUT ICYCLE
                END;    ! END OF CKPR

! CHECK ROUTINE FOR READ PROTECTION

        CKROPR:=
                BEGIN
                IF STKEYS[MAR<0:12>]<0> => CKPR
                END;    ! END OF CKROPR

! CHECK ROUTINE FOR SSK & ISK INSTRUCTIONS

        KEYCK:=
                BEGIN
                PSCHK NEXT
                (IF REG[R2]<28:31> => INTVEC<2>←1; INTCDE←8 NEXT BAILOUT ICYCLE) NEXT
                (IF REG[R2]<8:20> GTR MAXKEY => INTVEC<2>←1; INTCDE←5 NEXT BAILOUT ICYCLE)
                END; ! END OF KEYCK


! CHECK ROUTINE FOR BYTE ADDRESSES

        CKBTAD:=
                BEGIN
                IF MAR GTR MAXB => INTCDE←5; INTVEC<2>←1 NEXT BAILOUT ICYCLE
                END;

! CHECK 1/2 WORD ADDRESS ROUTINE

        CKHWAD:=
                BEGIN
                (IF MAR<23> => INTCDE←6; INTVEC<2>←1 NEXT BAILOUT ICYCLE) NEXT
                (IF MAR<0:22> GTR MAXH => INTCDE←5; INTVEC<2>←1 NEXT BAILOUT ICYCLE)
                END; ! END OF CKHWAD

! CHECK WORD ADDRESS ROUTINE

        CKWDAD:=
                BEGIN
                (IF MAR<22:23> => INTCDE←6; INTVEC<2>←1 NEXT BAILOUT ICYCLE) NEXT
                (IF MAR<0:21> GTR MAXW => INTCDE←5; INTVEC<2>←1 NEXT BAILOUT ICYCLE)
                END; ! END OF CKWDAD

! CHECK DOUBLE WORD ADDRESS ROUTINE

        CKDWAD:=
                BEGIN
                (IF MAR<21:23> => INTCDE←6; INTVEC<2>←1 NEXT BAILOUT ICYCLE) NEXT
                (IF MAR<0:20> GTR MAXDW => INTCDE←5; INTVEC<2>←1 NEXT BAILOUT ICYCLE)
                END; ! END OF CKDWAD

! READ A BYTE ROUTINE

        RDBYTE:=
```

```
S360 ISP DESCRIPTION

                BEGIN
                CKBTAD NEXT
                CKRDPR NEXT
                MBR<24:31>←MEMB[MAR]
                END; ! END OF RDBYTE

! WRITE A BYTE ROUTINE

        WRBYTE:=
                BEGIN
                CKBTAD NEXT
                CKPR NEXT
                MEMB[MAR]←MBR<24:31>
                END; ! END OF WRBYTE

! READ A 1/2 WORD ROUTINE

        READHW:=
                BEGIN
                CKHWAD NEXT
                CKRDPR NEXT
                MBR<16:31>←MEMH[MAR<0:22>]
                END;  ! END OF READHW
```

```
S360 ISP DESCRIPTION

! WRITE A 1/2 WORD ROUTINE

        WRHW:=
                BEGIN
                CKHWAD NEXT
                CKPR NEXT
                MEMH[MAR<0:22>]←MBR<16:31>
                END;  ! END OF WRHW

! READ A WORD ROUTINE

        READWD:=
                BEGIN
                CKWDAD NEXT
                CKRDPR NEXT
                MBR←MEMW[MAR<0:21>]
                END;  ! END OF READWD

! WRITE A WORD ROUTINE

        WRWD:=
                BEGIN
                CKWDAD NEXT
                CKPR NEXT
                MEMW[MAR<0:21>]←MBR
                END;  ! END OF WRWD

! OPERAND ONE ADDRESSING FOR SS

        ADBYT1:=
                BEGIN
                MAR←(AMAR1+LAUX1)<23:0>
                END;

! OPERAND TWO ADDRESSING FOR SS

        ADBYT2:=
                BEGIN
                MAR←(AMAR2+LAUX2)<23:0>
                END;


! FETCH OF L2 OPERAND IF POSSIBLE OR A LOAD OF ZERO INTO
!  THE MBR IF L2 FIELD IS EXHAUSTED

        L2FCH:=
                BEGIN
                DECODE (LAUX2 EQL 0) =>
                \0      BEGIN
                        LAUX2←(LAUX2 MINUS 1)<7:0> NEXT
                        ADBYT2 NEXT RDBYTE
                        END;
                \1      MBR←0
                END; ! END OF L2FCH


! DEVICE ADDRESSING FOR I/O INSTRUCTIONS

        ADRIO:=
                BEGIN
                CHAREG←(D1+REG[B1])<15:8>;
                DEVREG←(D1+REG[B1])<7:0>
                END;


! SIGN EXTENSION HALFWORD TO FULLWORD IN MBR

        SGNEXT:=
                BEGIN
                DECODE MBR<16>=>
                \0      MBR<0:15>←"0000;
                \1      MBR<0:15>←"FFFF
                END;  ! END OF SGNEXT
```

```
S360 ISP DESCRIPTION

! SET FIXED POINT CONDITION CODES

        SETFCC:=
                BEGIN
                CC←0 NEXT
                (DECODE REG[R1]<0>=>
                \0      (IF REG[R1]=>CC←2);
                \1      CC←1
                        ) NEXT
                (IF OVF => CC←3) NEXT
                (IF OVF AND FPOPMS => INTVEC<2>←1; INTCDE←0 NEXT BAILOUT ICYCLE)
                END;  ! END OF SETFCC

! ILLEGAL OP-CODE

        OPEX:=
                BEGIN
                INTCDE←1; INTVEC<2>←1 NEXT BAILOUT ICYCLE
                END;

! INSTRUCTION FETCH ROUTINE

        IFETCH:=
                BEGIN
                MAR←PC NEXT READHW NEXT
                IR<0:15>←MBR<16:31>;
                ILC←MBR<16>+MBR<17>+1;
                PC←(PC+(MBR<16>+MBR<17>+1)*2)<23:0>;
                OVF←0 NEXT
                (IF ILC GTR 1 =>
                        MAR←(MAR+2)<23:0> NEXT
                        READHW NEXT
                        IR<16:31>←MBR<16:31> NEXT
                        (IF ILC GTR 2 =>
                                MAR←(MAR+2)<23:0> NEXT
                                READHW NEXT
                                IR<32:47>←MBR<16:31>
                                ) ! END OF ILC GTR 2
                        ) ! END OF IF ILC GTR 1
                END;  ! END OF IFETCH
```

```
S360 ISP DESCRIPTION

!  PR INSTRUCTIONS

        SPM:=   ! SET PROGRAM MASK
                BEGIN
                PSW<34:39>←REG[R1]<2:7>
                END;    ! END OF SPM

        BALR:=  ! BRANCH AND LINK REGISTER
                BEGIN
                T24←REG[R2]<8:31>  NEXT
                REG[R1]←PSW<32:63> NEXT
                (IF R2 =>
                        BALR1:= (PC ← T24)
                        )
                END;  ! END OF BALR

        BCTR:=  ! BRANCH ON COUNTER REGISTER
                BEGIN
                T24 ← REG[R2]<8:31> NEXT
                REG[R1]←(REG[R1] MINUS 1)<31:0>  NEXT
                (IF REG[R1] =>
                        (IF R2 =>
                                BCTR1:= (PC←T24)
                                ) ! END OF IF R2
                        ) ! END OF OF REG[R1]
                END;  ! END OF BCTR

        BCR:=   ! BRANCH ON CONDITION REG
                BEGIN
                IF M1[CC] =>
                        (IF R2 =>
                                BCR1:=  (PC←REG[R2]<8:31>)
                                ) ! END OF IF R2
                END;

        SSK:=   ! SET STORAGE KEY (PROTECTION FEATURE INSTRUCTION)
                BEGIN
                KEYCK NEXT
                STKEYS[REG[R2]<8:20>]←REG[R1]<24:28>
                END;    ! END OF SSK

        ISK:=   ! INSERT STORAGE KEY (PROTECTION FEATURE INSTRUCTION)
                BEGIN
                KEYCK NEXT
                REG[R1]<24:28>←STKEYS[REG[R2]<8:20>] NEXT
                REG[R1]<29:31>←0
                END;    ! END OF ISK

        SVC:=   ! SUPERVISOR CALL
                BEGIN
                INTCDE← I2; INTVEC<1>←1 NEXT BAILOUT ICYCLE
                END;

        CLCL:=  ! COMPARE LOGICAL LONG (S/370)
                !
                ! THIS INSTRUCTION WAS ADDED FOR THE RUNNING OF
                ! BENCHMARK PROGRAMS.  IT IS NOT A TRUE DESCRIPTION
                ! OF THE INSTRUCTION SINCE IT IS NOT INTERRUPTABLE
                ! IN ITS PRESENT FORM.
                !
                BEGIN
                (IF (R1<3> OR R2<3>) => INTCDE←6; INTVEC<2>←1 NEXT BAILOUT ICYCLE) NEXT
                CLCLC1:= (CC←0) NEXT
                CLCL1:= BEGIN
                        IF (REG[R1+1]<8:31> NEQ 0) OR (REG[R2+1]<8:31> NEQ 0) =>
                                (DECODE (REG[R1+1]<8:31> NEQ 0) =>
                                \0      T8←REG[R1+1]<0:7>;
                                \1      BEGIN
                                        MAR←REG[R1]<8:31> NEXT
                                        RDBYTE NEXT
                                        T8←MBR<24:31>
                                        END     ! END OF \1
                                        ) NEXT  ! END OF DECODE
                                (DECODE (REG[R2+1]<8:31> NEQ 0) =>
```

```
                        \0      T8A←REG[R2+1]<0:7>;
                        \1      BEGIN
                                MAR←REG[R2]<8:31> NEXT
                                ROBYTE NEXT
                                T8A←MBR<24:31>
                                END     ! END OF \1
                                ) NEXT  ! END OF DECODE
                        CLCLC2:= BEGIN
                                DECODE T8 TST T8A =>
                                \LSS    CC←1;
                                \EQL    CC←0;
                                \GTR    CC←2
                                END NEXT        ! END OF CLCLC2
                        (IF CC EQL 0 =>
                                (IF REG[R1+1]<8:31> NEQ 0 =>
                                        REG[R1]←(REG[R1]+1)<23:0>;
                                        REG[R1+1]<8:31>←(REG[R1+1] MINUS 1)<23:0>
                                        ) NEXT  ! END OF IF REG[R1+1]
                                (IF REG[R2+1]<8:31> NEQ 0 =>
                                        REG[R2]←(REG[R2]+1)<23:0>;
                                        REG[R2+1]<8:31>←(REG[R2+1] MINUS 1)<23:0>
                                        ) NEXT  ! END OF IF REG[R2+1]
                                CLCL1)  ! END OF IF CC
                END     ! END OF CLCL1
END;    ! END OF CLCL
```

```
LPR:=   ! LOAD POSITIVE REGISTER
        BEGIN
        (DECODE REG[R2]<0> =>
        \0      REG[R1]←REG[R2];
        \1      REG[R1]←(MINUS REG[R2])<31:0>
                ) NEXT
        OVF←(REG[R1]  EQL "80000000) NEXT
        SETFCC
        END;    ! END OF LPR

LNR:=   ! LOAD NEGATIVE REGISTER
        BEGIN
        (DECODE REG[R2]<0> =>
        \0      REG[R1]←(MINUS REG[R2])<31:0>;
        \1      REG[R1]←REG[R2]
                ) NEXT
        SETFCC
        END;    ! END OF LNR

LTR:=   ! LOAD AND TEST REGISTER
        BEGIN
        REG[R1]←REG[R2] NEXT SETFCC
        END;

NR:=    ! AND REGISTER
        BEGIN
        REG[R1]←REG[R1] AND REG[R2]  NEXT
        NRCC:=  BEGIN
                CC←0 NEXT
                (IF REG[R1] => CC←1)
                END     ! END OF NRCC
        END;    ! END OF NR

CLR:=   ! COMPARE LOGICAL REGISTER
        BEGIN
        CLRCC:= BEGIN
                CC←0 NEXT
                (IF REG[R1] LSS REG[R2] =>
                        CC←1) NEXT
                (IF REG[R1] GTR REG[R2] =>
                        CC←2)
                END     ! END OF CLRCC
        END;    ! END OF CLR

OR.:=   ! OR REGISTER
        BEGIN
        REG[R1]←REG[R1] OR REG[R2]   NEXT
        ORCC:=  BEGIN
                CC←0 NEXT
                (IF REG[R1] => CC←1)
                END     ! END OR ORCC
        END;    ! END OF OR.

XR:=    ! EXCLUSIVE OR REGISTER
        BEGIN
        REG[R1]←REG[R1] XOR REG[R2]   NEXT
        XRCC:=  BEGIN
                CC←0 NEXT
                (IF REG[R1] => CC←1)
                END     ! END OR XRCC
        END;    ! END OF XR
```

```
LCR:=   ! LOAD AND COMPLEMENT REGISTER
        BEGIN
        REG[R1] ← (MINUS REG[R2])<31:0> NEXT
        OVF ← (REG[R1] EQL "80000000) NEXT
        SETFCC
        END;  ! END OF LCR

LR:=    ! LOAD REGISTER
        BEGIN
        REG[R1]←REG[R2]
        END;

CR:=    !COMPARE REGISTER
        BEGIN
        T33←(REG[R1] MINUS REG[R2]) NEXT
        CRCC:=  BEGIN
                CC←0 NEXT
                (IF T33 => CC←NOT  T33<1> + 1)
                END     ! END OF CRCC
        END; !END OF CR

AR:=    ! ADD REGISTER
        BEGIN
        T33←REG[R1]+REG[R2]    NEXT
        (DECODE REG[R1]<0>@REG[R2]<0> =>
        \00     OVF←(T33<0> NEQ T33<1>);
        \01     NOP;
        \10     NOP;
        \11     OVF←(T33<0> NEQ T33<1>)
                ) NEXT  ! END OF DECODE
        REG[R1]←T33<1:32> NEXT
        SETFCC
        END; ! END OF AR

SR:=    !SUBTRACT REGISTER
        BEGIN
        T33 ← ((NOT REG[R2]) + REG[R1] + 1)<32:0> NEXT
        (DECODE REG[R1]<0> @ REG[R2]<0> =>
        \00     NOP;
        \01     OVF ← (T33<0> NEQ T33<1>);
        \10     OVF ← (T33<0> NEQ T33<1>);
        \11     NOP
                ) NEXT !END OF DECODE
        REG[R1] ← T33<1:32> NEXT
        SETFCC
        END; ! END OF SR
```

```
MR:=    ! MULTIPLY REGISTER
        BEGIN
        (IF R1<3> => INTCDE←6; INTVEC<2>←1 NEXT BAILOUT ICYCLE) NEXT
        T1←(REG[R2]<0> XOR REG[R1+1]<0>) NEXT
        (DECODE REG[R2]<0> =>
        \0      T64←REG[R2];
        \1      T64←(MINUS REG[R2])<31:0>
                ) NEXT
        (DECODE REG[R1+1]<0> =>
        \0      T32←REG[R1+1];
        \1      T32←(MINUS REG[R1+1])<31:0>
                ) NEXT
        T64←T64<32:63>@T32 NEXT
        (IF T1 => T64←(MINUS T64)<63:0>) NEXT
        REG[R1]←T64<0:31>;
        REG[R1+1]←T64<32:63>
        END;    ! END OF MR

DR:=    !DIVIDE REGISTER
        BEGIN
        (IF R1<3> => INTCDE←6; INTVEC<2>←1 NEXT BAILOUT ICYCLE) NEXT
        T1←(REG[R1]<0> XOR REG[R2]<0>); T1A←REG[R1]<0>;
        T32←REG[R2] NEXT DIVREG←(REG[R1]@REG[R1+1]) NEXT
        (IF REG[R2]<0> => T32←(MINUS T32)<31:0>);
        (IF REG[R1]<0> => DIVREG←(MINUS DIVREG)<63:0>) NEXT
        (IF (DIVREG/T32)<63:31> => INTCDE←9; INTVEC<2>←1 NEXT BAILOUT ICYCLE) NEXT
        REG[R1+1]←(DIVREG/T32)<31:0> NEXT
        REG[R1]←(DIVREG MINUS (REG[R1+1]*T32))<31:0> NEXT
        (IF T1 => REG[R1+1]←(MINUS REG[R1+1])<31:0>) ;
        (IF T1A => REG[R1]←(MINUS REG[R1])<31:0>)
        END;    ! END OF DIVIDE REGISTER

ALR:=   !ADD LOGICAL
        BEGIN
        T33 ← REG[R1] + REG[R2] NEXT
        REG[R1] ← T33<1:32> NEXT
        ALRCC:= BEGIN
                CC ← T33<0> @ (T33<1:32> NEQ 0)
                END     ! END OF ALRCC
        END; ! END OF ALR

SLR:=   ! SUBTRACT LOGICAL REGISTER
        BEGIN
        T33 ← ((NOT REG[R2]) + REG[R1] + 1)<32:0> NEXT
        REG[R1] ← T33<1:32> NEXT
        SLRCC:= BEGIN
                CC←T33<0>@(T33<1:32> NEQ 0)
                END     ! END OF SLRCC
        END;    ! END OF SLR
```

```
        LPDR:=  ! LOAD POSITIVE (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF LDR

        LNDR:=  ! LOAD NEGATIVE (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF LNDR

        LTDR:=  ! LOAD AND TEST (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF LTDR

        LCDR:=  ! LOAD COMPLEMENT (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF LCDR

        HDR:=   ! HALVE (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF HDR

        LDR:=   ! LOAD (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF LDR

        CDR:=   ! COMPARE (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF CDR

        ADR:=   ! ADD NORMALIZED (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; !END OF ADR

        SDR:=   ! SUBTRACT NORMALIZED (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF SDR

        MDR:=   ! MULTIPLY (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF MDR

        DDR:=   ! DIVIDE (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF DDR

        AWR:=   ! ADD UNNORMALIZED (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF AWR

        SWR:=   ! SUBTRACT UNNORMALIZED (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF SWR

        LPER:=  ! LOAD POSITIVE (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF LPER

        LNER:=  ! LOAD NEGATIVE (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF LNER
```

```
LTER:=  ! LOAD AND TEST (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
        BEGIN
        NOP
        END; ! END OF LTER

LCER:=  ! LOAD COMPLEMENT (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
        BEGIN
        NOP
        END; ! END OF LCER

HER:=   ! HALVE (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
        BEGIN
        NOP
        END; ! END OF HER

LER:=   ! LOAD (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
        BEGIN
        NOP
        END; ! END OF LER

CER:=   ! COMPARE (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
        BEGIN
        NOP
        END; ! END OF CER

AER:=   ! ADD NORMALIZED (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
        BEGIN
        NOP
        END; ! END OF AER

SER:=   ! SUBTRACT NORMALIZED (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
        BEGIN
        NOP
        END; ! END OF SER

MER:=   ! MULTIPLY (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
        BEGIN
        NOP
        END; ! END OF MER

DER:=   ! DIVIDE (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
        BEGIN
        NOP
        END; ! END OF DER

AUR:=   ! ADD UNNORMALIZED (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
        BEGIN
        NOP
        END; ! END OF AUR

SUR:=   ! SUBTRACT UNNORMALIZED (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
        BEGIN
        NOP
        END; ! END OF SUR
```

```
S360 ISP DESCRIPTION

!  RR INSTRUCTION DECODE TABLE

        RR:=
                BEGIN
                (DECODE OPCODE<2:7> =>
                        OPEX;           ! 00
                        OPEX;           ! 01
                        OPEX;           ! 02
                        OPEX;           ! 03
                        SPM;            ! 04 SET PROGRAM MASK
                        BALR;           ! 05 BRANCH AND LINK
                        BCTR;           ! 06 BRANCH ON COUNT
                        BCR;            ! 07 BRANCH ON CONDITION
                        SSK;            ! 08 SET STORAGE KEY
                        ISK;            ! 09 INSERT STORAGE KEY
                        SVC;            ! 0A SUPERVISOR CALL
                        OPEX;           ! 0B
                        OPEX,           ! 0C
                        OPEX;           ! 0D
                        OPEX;           ! 0E
                        CLCL;           ! 0F COMPARE LOGICAL LONG (S/370)
                        LPR;            ! 10 LOAD POSITIVE
                        LNR;            ! 11 LOAD NEGATIVE
                        LTR;            ! 12 LOAD AND TEST
                        LCR;            ! 13 LOAD COMPLEMENT
                        NR;             ! 14 AND
                        CLR;            ! 15 COMPARE LOGICAL
                        OR;             ! 16 OR
                        XR;             ! 17 EXCLUSIVE OR
                        LR;             ! 18 LOAD
                        CR;             ! 19 COMPARE
                        AR;             ! 1A ADD
                        SR;             ! 1B SUBTRACT
                        MR;             ! 1C MULTIPLY
                        DR;             ! 1D DIVIDE
                        ALR;            ! 1E ADD LOGICAL
                        SLR;            ! 1F SUBTRACT LOGICAL
                        LPDR;           ! 20 LOAD POSITIVE (LONG)
                        LNDR;           ! 21 LOAD NEGATIVE (LONG)
                        LTDR;           ! 22 LOAD AND TEST (LONG)
                        LCDR;           ! 23 LOAD AND COMPLEMENT (LONG)
                        HDR;            ! 24 HALVE (LONG)
                        OPEX;           ! 25
                        OPEX;           ! 26
                        OPEX;           ! 27
                        LDR;            ! 28 LOAD (LONG)
                        CDR;            ! 29 COMPARE (LONG)
                        ADR;            ! 2A ADD NORMALIZED (LONG)
                        SDR;            ! 2B SUBTRACT NORMALIZED (LONG)
                        MDR;            ! 2C MULTIPLY (LONG)
                        DDR;            ! 2D DIVIDE (LONG)
                        AWR;            ! 2E ADD UNNORMALIZED (LONG)
                        SWR;            ! 2F SUBTRACT UNNORMALIZED (LONG)
                        LPER;           ! 30 LOAD POSITIVE (SHORT)
                        LNER;           ! 31 LOAD NEGATIVE (SHORT)
                        LTER;           ! 32 LOAD AND TEST (SHORT)
                        LCER;           ! 33 LOAD COMPLEMENT (SHORT)
                        HER;            ! 34 HALVE (SHORT)
                        OPEX;           ! 35
                        OPEX;           ! 36
                        OPEX;           ! 37
                        LER;            ! 38 LOAD (SHORT)
                        CER;            ! 39 COMPARE (SHORT)
                        AER;            ! 3A ADD NORMALIZED (SHORT)
                        SER;            ! 3B SUBTRACT NORMALIZED (SHORT)
                        MER;            ! 3C MULTIPLY (SHORT)
                        DER;            ! 3D DIVIDE (SHORT)
                        AUR;            ! 3E ADD UNNORMALIZED (SHORT)
                        SUR             ! 3F SUBTRACT UNNORMALIZED (SHORT)
                        )  ! END OF DECODE
                END;  ! END OF RR
```

```
S360 ISP DESCRIPTION

! RX INSTRUCTIONS

        STH:=   ! STORE HALFWORD
                BEGIN
                MBR←REG[R1]<16:31> NEXT WRHW
                END;

        LA:=    ! LOAD ADDRESS
                BEGIN
                REG[R1]←MAR
                END;

        STC:=   ! STORE CHARACTER
                BEGIN
                HIBYTE←REG[R1]<24:31> NEXT
                WRBYTE
                END;

        IC:=    ! INSERT CHARACTER
                BEGIN
                RDBYTE NEXT
                REG[R1]<24:31>←HIBYTE
                END; !END OF IC

        EX:=    ! EXECUTE
                BEGIN
                (IF EXRF => INTVEC<2>←1; INTCDE←3 NEXT BAILOUT ICYCLE) NEXT
                T4←R1; T2←0 NEXT
                EX1:=
                        BEGIN
                        READHW NEXT
                        IRW[T2]←MBR<16:31>  NEXT
                        T2←(T2+1)<1:0>; MAR←(MAR+2)<23:0>  NEXT
                        (IF (IR<0>+IR<1>) GEQ T2 => EX1) NEXT
                        (IF T4 => IR<8:15>←IR<8:15> OR REG[T4]<24:31>) NEXT
                        EXRF←1
                        END ! END OF EX1
                END;  ! END OF EX

        BAL:=   ! BRANCH AND LINK
                BEGIN
                REG[R1]←PSW<32:63> NEXT
                PC←MAR
                END;

        BCT:=   ! BRANCH ON COUNT
                BEGIN
                REG[R1]←(REG[R1] MINUS 1)<31:0> NEXT
                (IF REG[R1] =>
                        BCT1:=  (PC ← MAR)
                        ) ! END OF IF REG[R1]
                END;  ! END OF BCT
```

```
BC:=    ! BRANCH ON CONDITION
        BEGIN
        IF M1[CC] =>
                BC1:= (PC ← MAR)
        END;

LH:=    ! LOAD HALFWORD
        BEGIN
        READHW NEXT SGNEXT NEXT
        REG[R1]←MBR
        END; ! END OF LH

CH:=    !COMPARE HALFWORD
        BEGIN
        READHW NEXT SGNEXT NEXT
        T33 ← REG[R1] MINUS MBR NEXT
        CHCC:=  BEGIN
                CC←0 NEXT
                (IF T33 => CC←NOT  T33<1> + 1)
                END     ! END OF CHCC
        END; !END OF CH

AH:=    !ADD HALFWORD
        BEGIN
        READHW NEXT SGNEXT NEXT
        T33←REG[R1]+MBR    NEXT
        (DECODE REG[R1]<0>@MBR<0> =>
        \00     OVF←(T33<0> NEQ T33<1>);
        \01     NOP;
        \10     NOP;
        \11     OVF←(T33<0> NEQ T33<1>)
                ) NEXT  ! END OF DECODE
        REG[R1]←T33<1:32> NEXT
        SETFCC
        END; ! END OF AH

SH:=    !SUBTRACT HALFWORD
        BEGIN
        READHW NEXT SGNEXT NEXT
        T33 ← ((NOT MBR) + REG[R1] + 1)<32:0> NEXT
        (DECODE REG[R1]<0> @ MBR<0> =>
        \00     NOP;
        \01     OVF ← (T33<0> NEQ T33<1>);
        \10     OVF ← (T33<0> NEQ T33<1>);
        \11     NOP
                ) NEXT  ! END OF DECODE
        REG[R1] ← T33<1:32> NEXT
        SETFCC
        END; ! END OF SH
```

```
        MH:=    ! MULTIPLY HALFWORD
                ! MULTIPLIER IN MBR, MULTIPLICAND IN R1
                BEGIN
                READHW NEXT SGNEXT NEXT
                T1←(REG[R1]<0> XOR MBR<0>) NEXT
                (IF REG[R1]<0> => REG[R1]←(MINUS REG[R1])<31:0>);
                (IF MBR<0> => MBR←(MINUS MBR)<31:0>) NEXT
                REG[R1]←(REG[R1]*MBR)<31:0> NEXT
                (IF T1 => REG[R1]←(MINUS REG[R1])<31:0>)
                END; ! END OF MH

        CVD:=   ! CONVERT TO DECIMAL
                BEGIN
                CKDWAD NEXT
                MAR←(MAR+7)<23:0>;
                T1←REG[R1]<0>;
                T32←REG[R]] NEXT
                (IF T1 => T32←(MINUS T32)<31:0>) NEXT
                (DECODE ASCMSK =>
                \0      (DECODE T1 =>
                        \0      MBR<28:31>←'1100;
                        \1      MBR<28:31>←'1101
                                );
                \1      (DECODE T1 =>
                        \0      MBR<28:31>←'1010;
                        \1      MBR<28:31>←'1011
                                )
                        ) NEXT  ! END OF DECODE ASCMSK
                MBR<24:27>←(T32 MINUS ((T32/10)*10))<3:0> NEXT
                WRBYTE;
                T4←7 NEXT
                CVD1:=  BEGIN
                        IF T4 =>
                                MAR←(MAR MINUS 1)<23:0>;
                                T32←(T32/10)<31:0> NEXT
                                MBR<28:31>←(T32 MINUS ((T32/10)*10))<3:0> NEXT
                                T32←(T32/10)<31:0> NEXT
                                MBR<24:27>←(T32 MINUS ((T32/10)*10))<3:0> NEXT
                                WRBYTE;
                                T4←(T4 MINUS 1)<3:0> NEXT
                                CVD1
                        END     ! END OF CVD1
                END;  ! END OF CVD

        CVB:=   ! CONVERT TO BINARY
                BEGIN
                T64←0; T4←0; SCALE←10000000₀10000000; CKDWAD NEXT
                CVB1:=  BEGIN
                        RDBYTE NEXT
                        MAR←(MAR+1)<23:0>; T1←0;
                        DIGIT←MBR<24:27> NEXT
                        CVB2:=  BEGIN
                                (IF DIGIT GTR 9 =>
                                        INTVEC<2>←1; INTCODE←7 NEXT BAILOUT ICYCLE) NEXT ! END OF IF DIGIT
                                T64←(T64+(DIGIT*SCALE))<63:0> NEXT
                                (IF T4 LSS 14 =>
                                        SCALE←SCALE/10;
                                        T4←(T4+1)<3:0> NEXT
                                        (DECODE T1 EQL 0 =>
                                        \0      CVB1;
                                        \1      BEGIN
                                                T1←1;
                                                DIGIT←MBR<28:31> NEXT
                                                CVB2
                                                END ! END OF \1
                                                ) ! END OF DECODE
                                        ) NEXT ! END OF IF T4
                                DIGIT←MBR<28:31> NEXT   ! SIGN
                                (IF DIGIT LSS 10 =>
                                        INTVEC<2>←1;INTCODE←7 NEXT BAILOUT ICYCLE) NEXT
                                (IF T64<0:32> =>
                                        INTVEC<2>←1; INTCODE←9 NEXT BAILOUT ICYCLE) NEXT
                                (IF (DIGIT EQL '1011) OR (DIGIT EQL '1101) =>
                                        T64←(MINUS T64)<63:0>) NEXT
```

```
                        REG[R1]←T64<32:63>
                        END ! END OF CVB2
                END ! END OF CVB1
        END;  ! END OF CVB

ST:=    ! STORE
        BEGIN
        MBR←REG[R1]  NEXT
        WRWD
        END;  ! END OF ST

N:=     ! AND
        BEGIN
        READWD NEXT
        REG[R1]←REG[R1] AND MBR NEXT
        NCC:=   BEGIN
                CC←(REG[R1] NEQ 0)
                END     ! END OF NCC
        END;  ! END OF N

CL:=    ! COMPARE LOGICAL
        BEGIN
        READWD NEXT
        CLCC:=  BEGIN
                CC←0 NEXT
                (IF REG[R1] GTR MBR => CC←2) NEXT
                (IF REG[R1] LSS MBR => CC←1)
                END     ! END OF CLCC
        END;  ! END OF CL

O:=     ! OR
        BEGIN
        READWD NEXT
        REG[R1]←REG[R1] OR MBR NEXT
        OCC:=   BEGIN
                CC←(REG[R1] NEQ 0)
                END     ! END OF OCC
        END;  ! END OF O
```

```
X:=     ! EXCLUSIVE OR
        BEGIN
        READMD NEXT
        REG[R1]←REG[R1] XOR MBR NEXT
        XCC:=   BEGIN
                CC←(REG[R1] NEQ 0)
                END     ! END OF XCC
        END;  ! END OF X

L:=     ! LOAD
        BEGIN
        READMD NEXT
        REG[R1]←MBR
        END;  ! END OF L

C:=     ! COMPARE
        BEGIN
        READMD  NEXT
        T33←(REG[R1] MINUS MBR) NEXT
        CCC:=   BEGIN
                CC←0 NEXT
                (IF T33 => CC←NOT T33<1> + 1)
                END     ! END OF CCC
        END;  ! END OF C

A:=     ! ADD
        BEGIN
        READMD NEXT
        T33←REG[R1]+MBR NEXT
        (DECODE REG[R1]<0>@MBR<0> =>
        \00     OVF←(T33<0> NEQ T33<1>);
        \01     NOP;
        \10     NOP;
        \11     OVF←(T33<0> NEQ T33<1>)
                ) NEXT  ! END OF DECODE
        REG[R1]←T33<1:32> NEXT
        SETFCC
        END;  ! END OF A

S:=     ! SUBTRACT
        BEGIN
        READMD NEXT
        T33 ← ((NOT MBR) + REG[R1] + 1)<32:0> NEXT
        (DECODE REG[R1]<0> @ MBR<0> =>
        \00     NOP;
        \01     OVF←(T33<0> NEQ T33<1>);
        \10     OVF←(T33<0> NEQ T33<1>);
        \11     NOP
                ) NEXT  ! END OF DECODE
        REG[R1]←T33<1:32> NEXT
        SETFCC
        END;  ! END OF S
```

```
M:=     ! MULTIPLY
        BEGIN
        (IF R1<3> => INTCDE←6; INTVEC<2>←1 NEXT BAILOUT ICYCLE) NEXT
        READWD NEXT
        T1←(REG[R1+1]<0> XOR MBR<0>) NEXT
        (IF MBR<0> => MBR←(MINUS MBR)<31:0>);
        (DECODE REG[R1+1]<0> =>
        \0      T32←REG[R1+1];
        \1      T32←(MINUS REG[R1+1])<31:0>
                ) NEXT ! END OF DECODE
        T64←T32#MBR NEXT
        (IF T1 => T64←(MINUS T64)<63:0>) NEXT
        REG[R1]←T64<0:31>;
        REG[R1+1]←T64<32:63>
        END;  ! END OF M

D:=     ! DIVIDE
        BEGIN
        (IF R1<3> => INTCDE←6; INTVEC<2>←1 NEXT BAILOUT ICYCLE) NEXT
        READWD NEXT
        DIVREG←REG[R1]@REG[R1+1];
        T1←(REG[R1]<0> XOR MBR<0>) NEXT
        (IF DIVREG<0> =>
                DIVREG←(MINUS DIVREG)<63:0>);
        (IF MBR<0> =>
                MBR←(MINUS MBR)<31:0>) NEXT
        (IF (DIVREG/MBR)<63:31> => INTCDE←9; INTVEC<2>←1 NEXT BAILOUT ICYCLE) NEXT
        REG[R1+1]←(DIVREG/MBR)<31:0> NEXT
        (IF T1 => REG[R1+1]←(MINUS REG[R1+1])<31:0>) NEXT
        T1←REG[R1]<0> NEXT
        REG[R1]←(DIVREG MINUS(REG[R1+1]#MBR))<31:0> NEXT
        (IF T1 => REG[R1]←(MINUS REG[R1])<31:0>)
        END;  ! END OF D

AL:=    ! ADD LOGICAL
        BEGIN
        READWD NEXT
        T33←REG[R1]+MBR NEXT
        REG[R1] ← T33<1:32> NEXT
        ALCC:=  BEGIN
                CC←T33<0>@(T33<1:32> NEQ 0)
                END     ! END OF ALCC
        END;  ! END OF AL

SL:=    ! SUBTRACT LOGICAL
        BEGIN
        READWD NEXT
        T33 ← ((NOT MBR) + REG[R1] + 1)<32:0> NEXT
        REG[R1] ← T33<1:32> NEXT
        SLCC:=  BEGIN
                CC←T33<0>@(T33<1:32> NEQ 0)
                END     ! END OF SLCC
        END;  ! END OF SL
```

```
        STD:=   ! STORE (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF STD

        LD:=    ! LOAD (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF LD

        CD:=    ! COMPARE (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF CD

        AD:=    ! ADD NORMALIZED (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF AD

        SD:=    ! SUBTRACT NORMALIZED (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF SD

        MD:=    ! MULTIPLY (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF MD

        DD:=    ! DIVIDE (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF DD

        AW:=    ! ADD UNNORMALIZED (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF AW

        SW:=    ! SUBTRACT UNNORMALIZED (LONG) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF SW

        STE:=   ! STORE (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF STE

        LE:=    ! LOAD (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF LE

        CE:=    ! COMPARE (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF CE

        AE:=    ! ADD NORMALIZED (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF AE

        SE:=    ! SUBTRACT NORMALIZED (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF SE

        ME:=    ! MULTIPLY (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF ME
```

```
DE:=    ! DIVIDE (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
        BEGIN
        NOP
        END; ! END OF DE

AU:=    ! ADD UNNORMALIZED (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
        BEGIN
        NOP
        END; ! END OF AU

SU:=    ! SUBTRACT UNNORMALIZED (SHORT) (FLOATING-POINT FEATURE INSTRUCTION)
        BEGIN
        NOP
        END; ! END OF SU
```

```
S360 ISP DESCRIPTION

! RX INSTRUCTION DECODE TABLE

        RX:=
                BEGIN
                MAR←D1  NEXT    ! EFFECTIVE ADDRESS CALCULATION FOR RX
                (DECODE (B1 NEQ 0)@(X2 NEQ 0)=>
                \00     RX0000:=        (NOP);
                \01     RX00X2:=        (NOP);
                \10     RXB100:=        (NOP);
                \11     RXB1X2:=        (NOP)
                ) NEXT
                (IF B1 => MAR←(MAR+REG(B1))<23:0>) NEXT
                (IF X2 => MAR←(MAR+REG(X2))<23:0>) NEXT

                (DECODE OPCODE<2:7> =>          ! OPCODE DECODE FOR RX
                        STH;            ! 40 STORE HALFWORD
                        LA;             ! 41 LOAD ADDRESS
                        STC;            ! 42 STORE CHARACTER
                        IC;             ! 43 INSERT CHARACTER
                        EX;             ! 44 EXECUTE
                        BAL;            ! 45 BRANCH AND LINK
                        BCT;            ! 46 BRANCH ON COUNT
                        BC;             ! 47 BRANCH ON CONDITION
                        LH;             ! 48 LOAD HALFWORD
                        CH;             ! 49 COMPARE HALFWORD
                        AH;             ! 4A ADD HALFWORD
                        SH;             ! 4B SUBTRACT HALFWORD
                        MH;             ! 4C MULTIPLY HALFWORD
                        OPEX;           ! 4D
                        CVD;            ! 4E CONVERT TO DECIMAL
                        CVB;            ! 4F CONVERT TO BINARY
                        ST;             ! 50 STORE
                        OPEX;           ! 51
                        OPEX;           ! 52
                        OPEX;           ! 53
                        N;              ! 54 AND
                        CL;             ! 55 COMPARE LOGICAL
                        O;              ! 56 OR
                        X;              ! 57 EXCLUSIVE OR
                        L;              ! 58 LOAD
                        C;              ! 59 COMPARE
                        A;              ! 5A ADD
                        S;              ! 5B SUBTRACT
                        M;              ! 5C MULTIPLY
                        D;              ! 5D DIVIDE
                        AL;             ! 5E ADD LOGICAL
                        SL;             ! 5F SUBTRACT LOGICAL
                        STD;            ! 60 STORE (LONG)
                        OPEX;           ! 61
                        OPEX;           ! 62
                        OPEX;           ! 63
                        OPEX;           ! 64
                        OPEX;           ! 65
                        OPEX;           ! 66
                        OPEX;           ! 67
                        LD;             ! 68 LOAD (LONG)
                        CD;             ! 69 COMPARE (LONG)
                        AD;             ! 6A ADD NORMALIZED (LONG)
                        SD;             ! 6B SUBTRACT NORMALIZED (LONG)
                        MD;             ! 6C MULTIPLY (LONG)
                        DD;             ! 6D DIVIDE (LONG)
                        AW;             ! 6E ADD UNNORMALIZED (LONG)
                        SW;             ! 6F SUBTRACT UNNORMALIZED (LONG)
                        STE;            ! 70 STORE (SHORT)
                        OPEX;           ! 71
                        OPEX;           ! 72
                        OPEX;           ! 73
                        OPEX;           ! 74
                        OPEX;           ! 75
                        OPEX;           ! 76
                        OPEX;           ! 77
                        LE;             ! 78 LOAD (SHORT)
                        CE;             ! 79 COMPARE (SHORT)
                        AE;             ! 7A ADD NORMALIZED (SHORT)
```

```
                SE;             ! 7B SUBTRACT NORMALIZED (SHORT)
                ME;             ! 7C MULTIPLY (SHORT)
                DE;             ! 7D DIVIDE (SHORT)
                AU;             ! 7E ADD UNNORMALIZED (SHORT)
                SU              ! 7F SUBTRACT UNNORMALIZED (SHORT)
                )   ! END OF DECODE
        END;    ! END OF RX
```

```
S360 ISP DESCRIPTION

! RS,SI INSTRUCTIONS

        SSM:=   ! SET SYSTEM MASK
                BEGIN
                PSCHK NEXT
                RDBYTE NEXT
                PSW<0:7>←HIBYTE
                END; ! END OF SSM

        LPSW:=  ! LOAD PSW (PRIVILEGED INSTRUCTION)
                BEGIN
                PSCHK NEXT
                CKDWAD NEXT READWD NEXT
                PSW<0:15>←MBR<0:15>; MAR←(MAR+4)<23:0> NEXT
                READWD NEXT
                PSW<34:63>←MBR<2:31>
                END; ! END OF LPSW

        DIAG:=  ! DIAGNOSE (PRIVILEGED INSTRUCTION)
                !
                ! THIS INSTRUCTION DESCRIPTION DOES NOT CORRESPOND
                ! TO ANY PARTICULAR MODEL OF THE S/360 LINE.
                ! IT HAS BEEN MODIFIED FOR USE IN ENDING A
                ! SIMULATION RUN.
                !
                BEGIN
                PSCHK NEXT
                T2←ILC; T16←INTCDE NEXT
                SCNOUT←PSW NEXT
                PSW←MKNPSW NEXT
                INTCDE←T16; ILC←T2 NEXT
                STOPBIT←1   ! THIS WILL HALT MACHINE AND SIMULATION
                END; ! END OF DIAGNOSE

        WRD:=   ! WRITE DIRECT (DIRECT CONTROL FEATURE INSTRUCTION)
                BEGIN
                PSCHK NEXT RDBYTE NEXT
                SIGOUT←I2; IODREG←MBR<24:31>
                END; ! END OF WRD

        RDD:=   ! READ DIRECT (DIRECT CONTROL FEATURE INSTRUCTION)
                BEGIN
                PSCHK NEXT
                SIGOUT←I2; MBR<24:31>←IODREG NEXT
                WRBYTE
                END; ! END OF RDD

        BXH:=   ! BRANCH ON INDEX HIGH
                BEGIN
                (DECODE R3<3> =>
                \0      T32←REG[R3+1];
                \1      T32←REG[R3]
                        )   NEXT
                REG[R1]←(REG[R1]+REG[R3])<31:0> NEXT
                (IF (T32 MINUS REG[R1])<31> =>
                        BXH1:=  (PC←MAR)
                        )
                END; ! END OF BXH
```

```
BXLE:=  ! BRANCH ON INDEX LESS THAN OR EQUAL
        BEGIN
        (DECODE R3<3> =>
        \0      T32←REG[R3+1];
        \1      T32←REG[R3]
                )   NEXT
        REG[R1]←(REG[R1]+REG[R3])<31:0> NEXT
        (IF NOT((T32 MINUS REG[R1])<31>) =>
                BXLE1:= (PC ← MAR)
                )
        END; ! END OF BXLE

SRL:=   ! SHIFT RIGHT LOGICAL
        BEGIN
        REG[R1]←REG[R1]  ↑SR0  MAR<18:23>
        END; ! END OF SRL

SLL:=   ! SHIFT LEFT LOGICAL
        BEGIN
        REG[R1]←REG[R1]  ↑SL0  MAR<18:23>
        END; ! END OF SLL

SRA:=   ! SHIFT RIGHT SINGLE ARITHMETIC
        BEGIN
        (DECODE REG[R1]<0> =>
        \0      REG[R1] ← REG[R1] ↑SR0 MAR<18:23>;      ! POSITIVE
        \1      REG[R1] ← REG[R1] ↑SR1 MAR<18:23>       ! NEGATIVE
                ) NEXT  ! END OF DECODE
        SETFCC
        END;    ! END OF SRA

SLA:=   ! SHIFT LEFT SINGLE ARITHMETIC
        BEGIN
        T6←MAR<18:23> NEXT
        SLA1:=  (IF T6 =>
                        (IF REG[R1]<0> NEQ REG[R1]<1> => OVF←1) NEXT
                        REG[R1]<1:31>←(REG[R1]<1:31> ↑SL0 1);
                        T6←(T6 MINUS 1)<5:0> NEXT
                        SLA1
                ) NEXT
        SETFCC
        END; ! END OF SLA

SRDL:=  ! SHIFT RIGHT DOUBLE LOGICAL
        BEGIN
        (IF R1<3> => INTCDE←6; INTVEC<2>←1 NEXT BAILOUT ICYCLE) NEXT
        T64←REG[R1]@REG[R1+1] NEXT
        T64←T64  ↑SR0  MAR<18:23> NEXT
        REG[R1]←T64<0:31>;
        REG[R1+1]←T64<32:63>
        END; ! END OF SRDL
```

```
SLDL:=  ! SHIFT LEFT DOUBLE LOGICAL
        BEGIN
        (IF R1<3> => INTCOE←6; INTVEC<2>←1 NEXT BAILOUT ICYCLE) NEXT
        T64←REG[R1]@REG[R1+1] NEXT
        T64←T64  ↑SL0  MAR<18:23> NEXT
        REG[R1]←T64<0:31>;
        REG[R1+1]←T64<32:63>
        END; ! END OF SLDL

SRDA:=  ! SHIFT RIGHT DOUBLE ARITHMETIC
        BEGIN
        (IF R1<3> => INTCOE←6; INTVEC<2>←1 NEXT BAILOUT ICYCLE) NEXT
        (IF MAR<18:23> =>
                T64←REG[R1]@REG[R1+1] NEXT
                (DECODE T64<0> =>
                \0      T64←(T64 ↑SR0 MAR<18:23>);
                \1      T64←(T64 ↑SR1 MAR<18:23>)
                        ) NEXT ! END OF DECODE
                REG[R1]←T64<0:31>;
                REG[R1+1]←T64<32:63>
                ) NEXT
        SETFCC
        END; ! END OF SRDA

SLDA:=  ! SHIFT LEFT DOUBLE ARITHMETIC
        BEGIN
        (IF R1<3> => INTCOE←6; INTVEC<2>←1 NEXT BAILOUT ICYCLE) NEXT
        T6←MAR<18:23> NEXT
        T64←REG[R1]@REG[R1+1] NEXT
        SLDA1:= (IF T6 =>
                        (IF T64<0> NEQ T64<1> => OVF←1) NEXT
                        T64<1:63>←(T64<1:63> ↑SL0 1);
                        T6←(T6 MINUS 1)<5:0> NEXT
                        SLDA1
                        ) NEXT  ! END OF SLDA1
        REG[R1]←T64<0:31>;
        REG[R1+1]←T64<32:63> NEXT
        SETFCC
        END; ! END OF SLDA
```

```
        STM:=   ! STORE MULTIPLE
                BEGIN
                T4←R1 NEXT
                STM1:=  BEGIN
                        MBR←REG[T4] NEXT
                        WRWD NEXT
                        (IF T4 NEQ R3 =>
                                T4←(T4+1)<3:0>; MAR←(MAR+4)<23:0> NEXT
                                STM1
                                )
                        END
                END; ! END OF STM

        TM:=    ! TEST UNDER MASK
                BEGIN
                RDBYTE NEXT
                HIBYTE←I2 AND HIBYTE NEXT
                TMCC:=  BEGIN
                        CC←0 NEXT
                        (IF HIBYTE =>
                                CC←3 NEXT
                                (IF I2 XOR HIBYTE => CC←1)
                                ) ! END OF IF
                        END     ! END OF TMCC
                END; ! END OF TM

        MVI:=   ! MOVE IMMEDIATE
                BEGIN
                HIBYTE ← I2 NEXT
                WRBYTE
                END; ! END OF MOVE IMMEDIATE

        TS:=    ! TEST AND SET
                BEGIN
                NOP
                END; ! END OF TS

        NI:=    ! AND IMMEDIATE
                BEGIN
                RDBYTE NEXT
                HIBYTE ← (HIBYTE AND I2) NEXT
                WRBYTE NEXT
                NICC:=  BEGIN
                        CC←0 NEXT
                        (IF HIBYTE => CC←1)
                        END     ! END OF NICC
                END; ! END OF NI

        CLI:=   !COMPARE LOGICAL IMMEDIATE
                BEGIN
                RDBYTE NEXT
                CLICC:= BEGIN
                        CC←0 NEXT
                        (IF HIBYTE LSS I2 => CC←1) NEXT
                        (IF HIBYTE GTR I2 => CC←2)
                        END     ! END OF CLICC
                END; ! END OF CLI
```

S360 ISP DESCRIPTION

```
OI:=    ! OR IMMEDIATE
        BEGIN
        RDBYTE NEXT
        HIBYTE ← (HIBYTE OR I2) NEXT
        WRBYTE NEXT
        OICC:=  BEGIN
                CC←0 NEXT
                (IF HIBYTE => CC←1)
                END     ! END OF OICC
        END; ! END OF OI


XI:=    ! EXCLUSIVE OR IMMEDIATE
        BEGIN
        RDBYTE NEXT
        HIBYTE ← (HIBYTE XOR I2) NEXT
        WRBYTE NEXT
        XICC:=  BEGIN
                CC←0 NEXT
                (IF HIBYTE => CC←1)
                END     ! END OF XICC
        END; ! END OF XI

LM:=    ! LOAD MULTIPLE
        BEGIN
        T4←R1 NEXT
        LM1:=   BEGIN
                READWD NEXT
                REG(T4)←MBR NEXT
                (IF T4 NEQ R3 =>
                        T4←(T4+1)<3:0>; MAR←(MAR+4)<23:0> NEXT
                        LM1
                        ) !END OF IF T4
                END ! END OF LM1
        END; ! END OF LM
```

```
S360 ISP DESCRIPTION

! I/O INSTRUCTIONS
!
! FORMAT IS AS FOLLOWS:
!               BITS 0:7        OPCODE
!               BITS 8:15       UNUSED
!               BITS 16:19      BASE B1
!               BITS 20:31      DISPLACEMENT D1
!
!       THE SUM OF B1 AND D1 HAS THE FOLLOWING FORMAT:
!               BITS 0:15       UNUSED
!               BITS 16:23      CHANNEL ADDRESS*
!                               (0 IS MULTIPLEXOR)
!               BITS 24:31      DEVICE AND SUBCHANNEL ADDRESS
!
!                               * NOTE: ONLY CHANNELS 0-6 ARE VALID

        CHWAIT:= ! CHANNEL WAIT ROUTINE
                BEGIN
                IF NOT CHRLS => CHWAIT
                END; ! END OF CHANNEL WAIT

        CHINIT:= !
                BEGIN
                PSCHK NEXT
                ADRIO NEXT
                CHINST[IR<6:7>]←1;
                CHSEL←1 NEXT
                CHWAIT NEXT
                CC←CHANCC; CHSEL←0; CHINST[IR<6:7>]←0
                END; ! END OF CHINIT

        SIO:=   ! START I/O
                BEGIN
                CHINIT
                END; ! END OF SIO

        TIO:=   ! TEST I/O
                BEGIN
                CHINIT
                END; ! END OF TEST I/O

        HIO:=   ! HALT I/O
                BEGIN
                CHINIT
                END; ! END OF HALT I/O

        TCH:=   ! TEST CHANNEL
                BEGIN
                CHINIT
                END; ! END OF TCH
```

```
S360 ISP DESCRIPTION

!RS,SI INSTRUCTION DECODE TABLE

        RSSI:=
                BEGIN

                MAR ← D1 NEXT                   ! EFFECTIVE ADDRESS CALCULATION
                (IF B1 => RSSIB1:=( MAR ← (MAR+REG[B1])<23:0>)) NEXT

                (DECODE OPCODE<2:7> =>                  ! OPCODE DECODING
                        SSM;                    ! 80 SET SYSTEM MASK
                        OPEX;                   ! 81
                        LPSW;                   ! 82 LOAD PSW
                        DIAG;                   ! 83 DIAGNOSE
                        WRD;                    ! 84 WRITE DIRECT
                        RDD;                    ! 85 READ DIRECT
                        BXH;                    ! 86 BRANCH ON INDEX HIGH
                        BXLE;                   ! 87 BRANCH ON INDEX LESS THAN OR EQUAL
                        SRL;                    ! 88 SHIFT RIGHT LOGICAL
                        SLL;                    ! 89 SHIFT LEFT LOGICAL
                        SRA;                    ! 8A SHIFT RIGHT SINGLE ARITHMETIC
                        SLA;                    ! 8B SHIFT LEFT SINGLE ARITHMETIC
                        SRDL;                   ! 8C SHIFT RIGHT DOUBLE LOGICAL
                        SLDL;                   ! 8D SHIFT LEFT DOUBLE LOGICAL
                        SRDA;                   ! 8E SHIFT RIGHT DOUBLE ARITHMETIC
                        SLDA;                   ! 8F SHIFT LEFT DOUBLE ARITHMETIC
                        STM;                    ! 90 STORE MULTIPLE
                        TM;                     ! 91 TEST UNDER MASK
                        MVI;                    ! 92 MOVE IMMEDIATE
                        TS;                     ! 93 TEST AND SET
                        NI;                     ! 94 AND IMMEDIATE
                        CLI;                    ! 95 COMPARE LOGICAL IMMEDIATE
                        OI;                     ! 96 OR IMMEDIATE
                        XI;                     ! 97 EXCLUSIVE OR IMMEDIATE
                        LM;                     ! 98 LOAD MULTIPLE
                        OPEX;                   ! 99
                        OPEX;                   ! 9A
                        OPEX;                   ! 9B
                        SIO;                    ! 9C START I/O
                        TIO;                    ! 9D TEST I/O
                        HIO;                    ! 9E HALT I/O
                        TCH;                    ! 9F TEST CHANNEL
                        OPEX;                   ! A0
                        OPEX;                   ! A1
                        OPEX;                   ! A2
                        OPEX;                   ! A3
                        OPEX;                   ! A4
                        OPEX;                   ! A5
                        OPEX;                   ! A6
                        OPEX;                   ! A7
                        OPEX;                   ! A8
                        OPEX;                   ! A9
                        OPEX;                   ! AA
                        OPEX;                   ! AB
                        OPEX;                   ! AC
                        OPEX;                   ! AD
                        OPEX;                   ! AE
                        OPEX;                   ! AF
                        OPEX;                   ! B0
                        OPEX;                   ! B1
                        OPEX;                   ! B2
                        OPEX;                   ! B3
                        OPEX;                   ! B4
                        OPEX;                   ! B5
                        OPEX;                   ! B6
                        OPEX;                   ! B7
                        OPEX;                   ! B8
                        OPEX;                   ! B9
                        OPEX;                   ! BA
                        OPEX;                   ! BB
                        OPEX;                   ! BC
                        OPEX;                   ! BD
                        OPEX;                   ! BE
                        OPEX                    ! BF
                        )   ! END OF DECODE
```

```
S360 ISP DESCRIPTION

                END;    ! END OF RSSI
```

```
S360 ISP DESCRIPTION

! SS INSTRUCTIONS
!       FOR ALL OF THESE INSTRUCTIONS AN ADDRESSING ERROR OR A PROTECTION ERROR
!  RESULTS IN TERMINATION OF THE INSTRUCTION.  ALL, PART OF, OR NONE OF THE
!  RESULT MAY BE STORED.  THEREFORE THE RESULTANT DATA IS UNPREDICTABLE AND THE
!  SETTING OF THE CONDITION CODE, IF CALLED FOR MAY BE UNPREDICTABLE.  IN GENERAL
!  THE RESULTS SHOULD NOT BE USED.

        MVN:=   ! MOVE NUMERICS
                BEGIN
                LAUX1←0; LAUX2←0 NEXT
                MVN1:=  BEGIN
                        ADBYT2 NEXT RDBYTE NEXT
                        ADBYT1; T4←MBR<28:31> NEXT
                        RDBYTE NEXT
                        MBR<28:31>←T4 NEXT
                        WRBYTE NEXT
                        (IF LFLD GTR LAUX2 =>
                                LAUX1←(LAUX1+1)<7:0>; LAUX2←(LAUX2+1)<7:0> NEXT
                                MVN1
                                ) ! END OF IF LFLD
                        END ! END OF MVN1
                END; ! END OF MVN

        MVC:=   ! MOVE CHARACTER
                BEGIN
                LAUX1←0; LAUX2←0 NEXT
                MVC1:=   BEGIN
                        ADBYT2 NEXT RDBYTE NEXT
                        ADBYT1 NEXT
                        WRBYTE NEXT
                        (IF LFLD GTR LAUX2 =>
                                LAUX1←(LAUX1+1)<7:0>;LAUX2←(LAUX2+1)<7:0> NEXT
                                MVC1
                                )
                        END
                END; ! END OF MVC

        MVZ:=   ! MOVE ZONES
                BEGIN
                LAUX1←0; LAUX2←0 NEXT
                MVZ1:=  BEGIN
                        ADBYT2 NEXT RDBYTE NEXT
                        ADBYT1; T4←MBR<24:27> NEXT
                        RDBYTE NEXT
                        MBR<24:27>←T4 NEXT
                        WRBYTE NEXT
                        (IF LFLD GTR LAUX2 =>
                                LAUX1←(LAUX1+1)<7:0>; LAUX2←(LAUX2+1)<7:0> NEXT
                                MVZ1
                                ) ! END OF IF LFLD
                        END ! END OF MVZ1
                END; ! END OF MOVE ZONES
```

```
NC:=    ! AND CHARACTER
        BEGIN
        LAUX1←0; LAUX2←0;
        NCCC1:= (CC←0) NEXT
        NC1:=   BEGIN
                ADBYT2 NEXT RDBYTE NEXT
                LOBYTE←HIBYTE;
                ADBYT1 NEXT RDBYTE NEXT
                HIBYTE←(LOBYTE AND HIBYTE) NEXT WRBYTE NEXT
                NCCC2:= (IF HIBYTE => CC←1) NEXT
                (IF LFLD GTR LAUX2 =>
                        LAUX1←(LAUX1+1)<7:0>; LAUX2←(LAUX2+1)<7:0> NEXT
                        NC1
                        )
                END
        END; ! END OF NC

CLC:=   ! COMPARE LOGICAL CHARACTER
        BEGIN
        LAUX1←0; LAUX2←0;
        CLCCC1:=        (CC←0) NEXT
        CLC1:=  BEGIN
                ADBYT1 NEXT RDBYTE NEXT
                LOBYTE←HIBYTE;
                ADBYT2 NEXT RDBYTE NEXT
                (IF LOBYTE EQL HIBYTE =>
                        (IF LFLD GTR LAUX2 =>
                                LAUX1←(LAUX1+1)<7:0>; LAUX2←(LAUX2+1)<7:0> NEXT
                                CLC1
                                )
                        ) NEXT
                CLCCC2:= BEGIN
                        (IF LOBYTE LSS HIBYTE => CC←1) NEXT
                        (IF LOBYTE GTR HIBYTE => CC←2)
                        END     ! END OF CLCCC2
                END     ! END OF CLC1
        END; ! END OF CLC

OC:=    ! OR CHARACTER
        BEGIN
        LAUX1←0; LAUX2←0;
        OCCC1:= (CC←0) NEXT
        OC1:=   BEGIN
                ADBYT2 NEXT RDBYTE NEXT
                LOBYTE←HIBYTE;
                ADBYT1 NEXT RDBYTE NEXT
                HIBYTE←(LOBYTE OR HIBYTE) NEXT WRBYTE NEXT
                OCCC2:= (IF HIBYTE => CC←1) NEXT
                (IF LFLD GTR LAUX2 =>
                        LAUX1←(LAUX1+1)<7:0>; LAUX2←(LAUX2+1)<7:0> NEXT
                        OC1
                        )
                END
        END; ! END OF OC
```

```
XC:=    ! EXCLUSIVE OR CHARACTER
        BEGIN
        LAUX1←0; LAUX2←0;
        XCCC1:= (CC←0) NEXT
        XC1:=   BEGIN
                ADBYT2 NEXT RDBYTE NEXT
                LOBYTE←HIBYTE;
                ADBYT1 NEXT RDBYTE NEXT
                HIBYTE←(LOBYTE XOR HIBYTE) NEXT WRBYTE NEXT
                XCCC2:= (IF HIBYTE => CC←1) NEXT
                (IF LFLD GTR LAUX2 =>
                        LAUX1←(LAUX1+1)<7:0>; LAUX2←(LAUX2+1)<7:0> NEXT
                        XC1
                        )
                END
        END; ! END OF XC

TR:=    ! TRANSLATE
        BEGIN
        LAUX1←0; LAUX2←0 NEXT
        TR1:=   BEGIN
                ADBYT1 NEXT RDBYTE NEXT
                MAR←(AMAR2+HIBYTE)<23:0> NEXT RDBYTE NEXT
                ADBYT1 NEXT WRBYTE NEXT
                (IF LFLD GTR LAUX2 =>
                        LAUX1←(LAUX1+1)<7:0>; LAUX2←(LAUX2+1)<7:0> NEXT
                        TR1
                        )
                END
        END; ! END OF TR

TRT:=   ! TRANSLATE AND TEST
        BEGIN
        LAUX1←0; LAUX2←0;
        TRTCC1:= (CC←0) NEXT
        TRT1:=  BEGIN
                ADBYT1 NEXT RDBYTE NEXT
                MAR←(AMAR2+HIBYTE)<23:0> NEXT RDBYTE NEXT
                (IF HIBYTE =>
                        REG[1]<8:31>←(AMAR1+LAUX1)<23:0>;
                        REG[2]<24:31>←HIBYTE NEXT
                        TRTCC2:= BEGIN
                                CC←1 NEXT
                                (IF LFLD EQL LAUX1 => CC←2)
                                END     ! END OF TRTCC2
                        ) NEXT
                (IF (LFLD GTR LAUX2) AND (HIBYTE EQL 0) =>
                        LAUX1←(LAUX1+1)<7:0>; LAUX2←(LAUX2+1)<7:0> NEXT
                        TRT1
                        )
                END
        END; ! END OF TRT
```

```
        ED:=    ! EDIT (DECIMAL FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF ED

        EDMK:=  ! EDIT AND MARK (DECIMAL FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF EDMK

        MVO:=   ! MOVE WITH OFFSET
                BEGIN
                LAUX1←L1; LAUX2←L2 NEXT
                ADBYT1 NEXT RDBYTE NEXT
                T4←MBR<28:31>; ADBYT2 NEXT RDBYTE NEXT
                MVO1:=  BEGIN
                        ADBYT1 NEXT
                        MBR←MBR 1SL0 4 NEXT
                        MBR<28:31>←T4 NEXT
                        WRBYTE; T4←MBR<24:27> NEXT
                        (IF LAUX1 NEQ 0 =>
                                LAUX1←(LAUX1 MINUS 1)<7:0> NEXT
                                L2FCH NEXT
                                MVO1
                                ) ! END OF IF LAUX1
                        END ! END OF MVO1
                END; ! END OF MVO

        PACK:=  ! PACK
                BEGIN
                LAUX1←L1; LAUX2←L2 NEXT
                ADBYT2 NEXT RDBYTE NEXT
                MBR<24:31>←MBR<28:31>@MBR<24:27> NEXT
                PACK1:= BEGIN
                        ADBYT1 NEXT
                        WRBYTE NEXT
                        (IF LAUX1 NEQ 0 =>
                                LAUX1←(LAUX1 MINUS 1)<7:0>;
                                L2FCH NEXT
                                T4←MBR<28:31> NEXT
                                L2FCH NEXT
                                MBR<24:31>←MBR<28:31>@T4 NEXT
                                PACK1
                                ) ! END OF IF LAUX1
                        END ! END OF PACK1
                END; ! END OF PACK

        UNPK:=  ! UNPACK
                BEGIN
                LAUX1←L1; LAUX2←L2;
                (DECODE ASCMSK=>
                \0      ZONE←'1111;
                \1      ZONE←'0101
                        ) NEXT
                ADBYT2 NEXT RDBYTE NEXT
                MBR<24:31>←MBR<28:31>@MBR<24:27> NEXT
                UNPK1:= BEGIN
                        ADBYT1 NEXT
                        WRBYTE NEXT
                        (IF LAUX1 NEQ 0 =>
                                LAUX1←(LAUX1 MINUS 1)<7:0>;
                                L2FCH NEXT
                                T4←MBR<24:27> NEXT
                                MBR<24:27>←ZONE; ADBYT1 NEXT
                                WRBYTE NEXT
                                (IF LAUX1 NEQ 0 =>
                                        LAUX1←(LAUX1 MINUS 1)<7:0>;
                                        MBR<24:31>←ZONE@T4 NEXT
                                        UNPK1
                                        ) ! END OF IF LAUX1
                                ) ! END OF IF LAUX1
                        END ! END OF UNPK1
                END; ! END OF UNPACK
```

```
        ZAP:=   ! ZERO AND ADD (DECIMAL FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF ZAP

        CP:=    ! COMPARE DECIMAL (DECIMAL FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF COMPARE DECIMAL

        AP:=    ! ADD DECIMAL (DECIMAL FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF ADD DECIMAL

        SP:=    ! SUBTRACT DECIMAL (DECIMAL FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF SUBTRACT DECIMAL

        MP:=    ! MULTIPLY DECIMAL (DECIMAL FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF MULTIPLY DECIMAL


        DP:=    ! DIVIDE DECIMAL (DECIMAL FEATURE INSTRUCTION)
                BEGIN
                NOP
                END; ! END OF DIVIDE DECIMAL
```

```
S360 ISP DESCRIPTION

! SS INSTRUCTION DECODE TABLE

        SS:=
                BEGIN

                AMAR1←D1;  AMAR2←D2  NEXT       ! EFFECTIVE ADDRESS CALCULATION
                (IF B1 => SSB1:= (AMAR1←(AMAR1+REG[B1])<23:0>));
                (IF B2 => SSB2:=(AMAR2←(AMAR2+REG[B2])<23:0>))  NEXT

                (DECODE OPCODE <2:7> =>          ! OPCODE DECODING
                        OPEX;           ! C0
                        OPEX;           ! C1
                        OPEX;           ! C2
                        OPEX;           ! C3
                        OPEX;           ! C4
                        OPEX;           ! C5
                        OPEX;           ! C6
                        OPEX;           ! C7
                        OPEX;           ! C8
                        OPEX;           ! C9
                        OPEX;           ! CA
                        OPEX;           ! CB
                        OPEX;           ! CC
                        OPEX;           ! CD
                        OPEX;           ! CE
                        OPEX;           ! CF
                        OPEX;           ! D0
                        MVN;            ! D1 MOVE NUMERICS
                        MVC;            ! D2 MOVE CHARACTER
                        MVZ;            ! D3 MOVE ZONES
                        NC;             ! D4 AND CHARACTER
                        CLC;            ! D5 COMPARE LOGICAL CHARACTER
                        OC;             ! D6 OR CHARACTER
                        XC;             ! D7 EXCLUSIVE OR CHARACTER
                        OPEX;           ! D8
                        OPEX;           ! D9
                        OPEX;           ! DA
                        OPEX;           ! DB
                        TR;             ! DC TRANSLATE
                        TRT;            ! DD TRANSLATE AND TEST
                        ED;             ! DE EDIT
                        EDMK;           ! DF EDIT AND MARK
                        OPEX;           ! E0
                        OPEX;           ! E1
                        OPEX;           ! E2
                        OPEX;           ! E3
                        OPEX;           ! E4
                        OPEX;           ! E5
                        OPEX;           ! E6
                        OPEX;           ! E7
                        OPEX;           ! E8
                        OPEX;           ! E9
                        OPEX;           ! EA
                        OPEX;           ! EB
                        OPEX;           ! EC
                        OPEX;           ! ED
                        OPEX;           ! EE
                        OPEX;           ! EF
                        OPEX;           ! F0
                        MVO;            ! F1 MOVE WITH OFFSET
                        PACK;           ! F2 PACK
                        UNPK;           ! F3 UNPACK
                        OPEX;           ! F4
                        OPEX;           ! F5
                        OPEX;           ! F6
                        OPEX;           ! F7
                        ZAP;            ! F8 ZERO AND ADD
                        CP;             ! F9 COMPARE DECIMAL
                        AP;             ! FA ADD DECIMAL
                        SP;             ! FB SUBTRACT DECIMAL
                        MP;             ! FC MULTIPLY PACKED
                        DP;             ! FD DIVIDE PACKED
                        OPEX;           ! FE
                        OPEX            ! FF
```

```
                    )   ! END OF DECODE
END;                ! END OF SS
```

```
S360 ISP DESCRIPTION

! INTERRUPT SERVICE ROUTINES

        INT:=
                BEGIN
                T2←ILC NEXT     ! SAVE INSTRUCTION LENGTH

                        ! HANDLE PRIORITY (1) INTERRUPTS

                (IF INTVEC<0> AND MCHKMK =>
                        MKOPSW←PSW NEXT
                        MKOPSW<16:31>←0 NEXT
                        SCNOUT←PSW NEXT
                        PSW←MKNPSW;
                        INTVEC<0:2>←0
                        ) NEXT

                        ! HANDLE PRIORITY (2) INTERRRUPTS

                (IF INTVEC<1> =>
                        SVCPSW←PSW NEXT
                        PSW←SVNPSW;
                        INTVEC<1>←0
                        ) NEXT

                (IF INTVEC<2> =>
                        PROPSW←PSW NEXT
                        PSW←PRNPSW;
                        INTVEC<2>←0
                        ) NEXT

                        ! HANDLE PRIORITY (3) INTERRUPTS

                (IF INTVEC<3> AND CHAMSK =>
                        INTCDE←EXTREG NEXT
                        EXOPSW←PSW NEXT
                        PSW←EXNPSW;
                        INTVEC<3>←0
                        ) NEXT

                        ! HANDLE PRIORITY (4) INTERRUPTS

                (IF INTVEC<4> AND IOMSK =>
                        INTCDE←DEVREG NEXT
                        IOOPSW←PSW NEXT
                        PSW←IONPSW;
                        INTVEC<4>←0
                        ) NEXT

                INTCDE←0; ILC←T2  ! RESET ILC & INTERRUPT CODE

                END; ! END OF INTERRUPT HANDLING
```

```
S360 ISP DESCRIPTION

! INSTRUCTION DECODING SECTION

        IEXEC:=
                BEGIN
                DECODE OPCODE<0:1> =>
                        RR;
                        RX;
                        RSSI;
                        SS
                END;  ! END OF IEXEC


        ICYCLE :=
                BEGIN
                IFETCH NEXT
                IEXEC NEXT
                (IF EXRF => IEXEC NEXT EXRF←0)
                END

        ERALCED  ! END OF DECLARATIONS

! MAIN EXECUTABLE PROGRAM

        RUN:=
                BEGIN
                (IF NOT STOPBIT =>
                        (IF NOT WAITST =>
                                ICYCLE
                                ) NEXT ! END OF NOT WAITST
                        INT NEXT
                        RUN
                        ) ! END OF NOT STOPBIT
                END ! END OF RUN LOOP

        ! END OF S360
        )
```

```
INTERDATA 8/32 ISP DESCRIPTION

INTERDATA := ( DECLARE

!                                                                       !
!       USEFUL  MACROS --- SYNONYMS                                     !
!                                                                       !
        MACRO   BEGIN :=($
        MACRO   END := )$
        MACRO   IFF := DECODE(NOT ($
        MACRO   THEN := ))=>$
        MACRO   ELSE := \ELSE $
!                                                                       !
!       INTERDATA STORAGE RESOURCES                                     !
!                                                                       !

REG[0:127]<0:31> ;      !8  SETS OF 16 REGISTERS
                        !WARNING: IMPLEMENTATION ASSUMES ONLY 8 REGISTER SETS

        macro maxbytes:="ffff$
BMEM[0:maxbytes]<0:7> ;                         !BYTE-ADDRESSABLE MEMORY
        HMEM[0:"7fff]<0:15> := BMEM[0:maxbytes]<0:7> ;  !HALF-WORDS
        WMEM[0:"3fff]<0:31> := BMEM[0:maxbytes]<0:7> ;      !FULL-WORDS
!
!       PROGRAM STATUS WORD < PSW > AND ITS SUBFIELDS
!
PSW<0:63> ;
!
\PSW.SUBFIELDS
\COND.CODE      CC<0:3> := PSW<28:31> ;         !CONDITION CODE
        C<> := PSW<28> ;                        !CARRY BIT
        V<> := PSW<29> ;                        !OVERFLOW BIT
        G<> := PSW<30> ;                        !GREATER-THAN BIT
        L<> := PSW<31> ;                        !LESS-THAN BIT

\REGISTER.SET   R<0:3> := PSW<24:27> ;          !CURRENT REGISTER SET
        LOC<0:19> := PSW<44:63> ;               !LOCATION COUNTER
\INTERRUPT.MASKBITS
MACRO   II := PSW<17>@PSW<20>$                  !IMMEDIATE INTERRUPTS MASK BITS
        I0<> := PSW<17> ;                       !HIGH-ORDER BIT OF II
        I1<> := PSW<20> ;                       !LOW-ORDER BIT OF II
        W<> := PSW<16> ;                        !WAIT-STATE BIT
        M<> := PSW<18> ;                        !MACHINE MALFUNCTION INTERRUPT MASK BIT
        A<> := PSW<19> ;                        !ARITHMETIC FAULT INTERRUPT MASK BIT
        P<> :=PSW<23> ;                         !PROTECT MODE INTERRUPT MASK BIT
        R.P<> := PSW<21> ;                      !MEMORY RELOCATION/PROTECTION VIOLATION MASK
        Q<> := PSW<22> ;                        !QUEUE SERVICE INTERRUPT MASK BIT
!
!       INSTRUCTION REGISTER <IR> AND ITS SUBFIELDS
!
IR<0:47> ;
!
\IR.SUBFIELDS
        OP<0:7> := IR<0:7> ;                    !OPCODE
        R1<0:3> := IR<8:11>;                    !FIRST OPERAND-REGISTER
        R2<0:3> := IR<12:15> ;                  !SECOND OPERAND-REGISTER
        N<0:3> := IR<12:15> ;                   !4-BIT LITERAL
        X2<0:3> := IR<12:15> ;                  !SECOND OPERAND INDEX REGISTER
        RI1<0:15> := IR<16:31> ;                !16-BIT CONSTANT
        RI2<0:31> := IR<16:47> ;                !32-BIT CONSTANT
        D2<0:13> := IR<18:31> ;                 !2'S COMPLEMENT DISPLACEMENT
        FX2<0:3> := IR<12:15> ;                 !FIRST INDEX REG
        SX2<0:3> := IR<20:23> ;                 !2ND INDEX REG
        A2<0:23> := IR<24:47> ;
        RXTYPE<0:1> := IR<16:17> ;   !USED TO DECODE RX TYPE
!
!       RR FORMAT SUBFIELDS: OP, R1, R2
!       SF FORMAT SUBFIELDS: OP, R1, N
!        RI FORMAT SUBFIELDS: OP, R1, X2, (RI1 OR RI1)
!       RX FORMAT SUBFIELDS: OP, R1, ((X2, D2) OR (FX1, FX2, A2))
!
```

```
INTERDATA 8/32 ISP DESCRIPTION

!               ISP TEMPORARY REGISTERS

        RUN<>;                          !RUN FLAG: 1---GO; 0---HALT
        NOPR<>;                         !NOP REGISTER
        MACRO NOP:=NOPR+0$              !NULL STATEMENT
        NBYTES<0:2>;                    !NO. OF BYTES TO READ/WRITE FROM/INTO MEMORY
                                !NBYTES CAN BE 1, 2, OR 4 (BYTE, HALF OR FULL WORD)
        INSTR<>;                        !FLAG: 0=> DATA FETCH; 1=> INSTR. FETCH
        r.w<>;                          !r.w = 0 -- read; r.w = 1 --- write
        fixed.float<>;                  !fixed.float = 0 --- fixed pt arithmetic; fixed.float = 1 --- floating pt
        intvec<0:3>;                    !interrupt vector (contains a 1 in the bit position corresponding to a pr
        intlev<0:3>;                    !contains the current interrupt level being processed
        devnum<0>;                      !dummy device number
        devstat<>;                      !dummy device status
        MACR<0:24>;                     !mac address translation register
macro wmacr:=macr ^sr0 2 $
macro hmacr:=macr ^sr0 1 $
        MAR<0:19> ;                     !MEMORY ADDRESS REGISTER
        MBR<0:31> ;                     !MEMORY BUFFER REGISTER
macro wmar:=mar ^sr0 2 $
macro hmar:=mar ^sr0 1 $
        SET<0:2>:=R<1:3> ;              !REGISTER SET SELECTION: 8 sets only
        EA<0:19> ;                      !EFFECTIVE ADDRESS FOR RX & RI FORMAT OPERANDS
        CCOP<0:31>;                     !COND.CODE OPERAND (32-BIT)
        RIOPND<0:31> ;                  !RI FORMAT OPERAND
        temp1<>;                        !1-bit temp reg
MACRO sign:=temp1$              !1-bit temp
        temp4<0:3>;                     !4-bit temp reg
        low<0:3>;                       !4-bit temp reg used by intchk
        temp5<0:4>;                     !5-bit temp reg
        temp8<0:7>;                     !8-bit temporary reg
        TEMP16<0:15>;                   !16-bit TEMPORARY REGISTER
        dat16<0:15> ;                   !another 16-bit temporary reg
        MACRO max16:=dat16$             !another 16-bit temporary register (used with list instructions)
        TEMP17<0:16>;                   !17-BIT TEMPORARY REGISTER
        TEMP20<0:19> ;                  !20-BIT TEMPORARY REGISTER
        TEMP32<0:31> ;                  !32-BIT TEMPORARY REGISTER
        dat32<0:31> ;                   !another 32-bit temporary reg
        MACRO div32:=dat32$             !another 32-bit temp (used in divide instructions
        TEMP33<0:32> ;                  !33-bit temporary register
        TEMP64<0:63> ;                  !64-BIT TEMPORARY REGISTER
        Q64<0:63> ;                     !another 64-bit temp (used in Divide instructions)
        OLDPSW<0:63> ;                  !USED TO HOLD CURRENT PSW
MACRO   NEWPSW:= TEMP20$                !ADDRESS OF NEW PSW

!
```

```
INTERDATA 8/32 ISP DESCRIPTION

        !ISP SUBROUTINES --- CALLED BY OTHER ROUTINES

        !MAC is called by MEMRD and MEMWT
        !performs actions of the Memory Access Controller


MAC:= begin                             !memory access control
        NOP                             !NOT CURRENTLY IMPLEMENTED IN THIS ISP
end;            !mac

        !BNDRYCHK is called by MEMRD and MEMWT
        !parameters: nbytes (readonly); mar (read/write)

BNDRYCHK:=Begin                 !check address boundary
                        !if data read => boundary error causes machine interrupt
                        !if instruction read => address are truncated to lower-most boundary
                        !initially, just truncate all addresses, since INTERDATA
                        !has not yet implemented the M int for boundary errors
        (IFF (nbytes EQL 2) THEN mar←mar AND "fffe ;
                        !halfword accesses must be on a halfword boundary
                ELSE
                (IF nbytes eql 4 => mar←mar AND "fffc )
                        !if fullword read, must be on fullword boundary
        )       !end IFF
END;    !end bndrychk


MEMRD := BEGIN                          !READ NBYTES FROM MEMORY ADDRESS MAR INTO MBR
        r.w←0 Next                      !doing a read
        bndrychk  Next                  !chk word boundary
         MACR←MAR Next                  !initialize macr
        (IF R.P => MAC  ) NEXT          ! memory access control
        (decode nbytes =>
        \0      nop;
        \1      mbr←bmem[macr];         !read byte zero-fill
        \2      mbr←hmem[hmacr];        !read halfword
        \3      nop;
        \4      mbr←wmem[wmacr]         !read word
        )       !end decode
END ; !MEMRD ;


MEMWT:=BEGIN                            !MEMORY WRITE ROUTINE
        BNDRYCHK  NEXT                  !CHECK FOR BOUNDARY ALIGNMENT
        R.W←1 ;MACR ←MAR  NEXT          !DOING A WRITE
        (IF R.P => MAC )  NEXT          !IF R.P => MEMORY TRANSLATE
        (DECODE NBYTES =>
        \0      NOP;
        \1      BMEM[MACR] ← MBR<24:31> ;       !BYTE WRITE
        \2      HMEM[HMACR]←MBR<16:31>;         !HALFWORD WRITE
        \3      NOP;
        \4      WMEM[WMACR]←MBR<0:31>           !FULLWORD WRITE
        )       !END DECODE
END;    !MEMWT


CCFIXED:=BEGIN                  !ccfixed sets CC using value of parameter ccop
                                !G and L set according to ccop value
                                !C and V set to 0
        (DECODE CCOP<0> =>      !test sign of ccop
         \0.NONNEG
                 (IFF (CCOP EQL 0) THEN CC←0 ;  !clear G and L
                ELSE CC←2       !set G
                );
         \1.NEG  CC←1           !set L
        ) ! END DECODE
END;    !CCFIXED


        !SYSINT : LOADS NEW PSW AND SAVES OLD PSW IN REGISTERS 14 AND 15 OF
        ! THE NEWLY SELECTED SET
        !SYSINT IS USED WHEN ANY OF THE 5 TYPES OF INTERRUPTS OCCUR:
        !SUPERVISOR CALL, ILLEGAL (OR PROTECTION) INSTRUCTION,
        !SYSTEM QUEUE SERVICE, ARITHMETIC FAULT .
```

```
INTERDATA 8/32 ISP DESCRIPTION

        !INPUT PARAMETERS:
        !NEWPSW - CONTAINS MEMORY ADDRESS OF NEW PSW

SYSINT := BEGIN
        OLDPSW←PSW  NEXT                    ! SAVE CURRENT PSW
        mar←newpsw  NEXT                    !ADDR OF NEW PSW
        TEMP64<0:31>←WMEM[WMAR] NEXT        !PART OF NEW PSW
        MAR←(MAR+4)<19:0>  NEXT
        TEMP64<32:63>←WMEM[WMAR] NEXT
        PSW←TEMP64 NEXT                     !NEW PSW LOADED
        !PSW←WMEM[NEWPSW]<0:31>@WMEM[NEWPSW + 4]<0:31>
        !       PSW LOADED WITH DATA IN MEMORY AT ADDRESS NEWPSW
        REG[SET@"E]←OLDPSW <0:31>;
        REG[SET@"F]←OLDPSW<32:63>
        !OLD PSW LOADED INTO REGS 14 AND 15 OF NEW SET
END ;   !SYSINT

QSCHK := BEGIN                              !SYSTEM QUEUE SERVICE CHECK
        mar←"80 Next                        !address of system queue is at location "80
        mbr←wmem[wmar] Next                 !fetch address of system queue
        ea←mbr<13:31> Next                          !low 28-bit address
        mar← (ea+2)<19:0>; nbytes←2 Next            !prepare to fetch 2nd halfword of list headr of system queue
        Mewrd Next                          !mbr now contains the no. of slots used in the list/queue
        (IF mbr =>                          !is there an entry in the queue
                newpsw←"88 Next             !if so, syst. queue interrupt occurs
                sysint Next                 !change processor state(for interrupt handling)
                reg[set@ "d]←ea             !place address of system queue in reg. 13 of new set
        )       !end IF
END ;   !QSCHK


        !ARITHCHK called by DIVIDE instructions
        !parameters: fixed.float (readonly)
        !ARITHCHK calls SYSINT


ARITHCHK:=Begin                             !arithmetic interrupt check
        (IF a => newpsw←"48;                !address of new psw for arith. ints handler
                sysint;                     !change processor state (ie. swap psws)
                (DECODE fixed.float =>      !was it a fixed pt  or floating pt arithmetic error
                \0.fixed        c←0;        !fixed pt operation => clear carry
                \1.float        c←1         !floating pt => set carry
                )       !end DECODE
        )       !end IF
End;    !arithchk


getdevinfo:=Begin                           !respond to i/o interrupt by getting device no. and status
        nop                                 !dummy parameters returned: devnum & devstat
        End;


        !IOINT is called by INTCHK when an interrupt of level intlev is to be processed
        !parameters: intlev (readonly)
        !IOINT calls GETDEVINFO

IOINT:=Begin                                !an i/o interrupt of level intlev exists (and is enabled)
                                            !process interrupt !!!
        oldpsw←psw Next
        psw<0:31>← ("28 @ intlev)<31:0> †SL0 4 Next     !set new psw status
                                            !change to the reg. set corresponding to intlev
        reg[set@("0<3:0>)]←oldpsw<0:31>;            !save old psw in registers
        reg[set@("1<3:0>)]←oldpsw<32:63> Next       !   1 and 2 of the new set
        getdevinfo Next                     !get device no. and status
        reg[set@("2<3:0>)]← devnum;                 !save device number in reg 2 of new set
        reg[set@("3<3:0>)]← devstat Next            !save device status in reg 3 of new set
        loc←("d0 + (('0@devnum)<19:0> †SL0 2))<19:0>    !get address of device handler from interrupt handler tat
End;    !ioint
```

```
INTERDATA 8/32 ISP DESCRIPTION

        !INSTRUCTION FORMAT ROUTINES

                !   rr & sf format instructions


rrformat := begin
        LOC ← (MAR +2)<19:0> ;          !UPDATE LOC
        instr←0                         !END of instruction fetch mode
END ;   !rrformat


        !sf instruction format


sfformat:=Begin
        rrformat                        !sfformat uses same routine as rrformat
End;    !sfformat


                !ri type 1 format instruction

ri1format:= begin
        mar←(mar+2)<19:0>  NEXT         !update mar
        memrd   NEXT                    !read nbytes more
        ir<16:31>←mbr<16:31>  NEXT
                !calculate ri operand

        (decode ri1<0> =>               !test sign bit
                riopnd←ri1 ;            !nonneg
                riopnd← ("ffff @ri1)<31:0>      !neg
        ) NEXT          !END decode
        (if x2 => ri1ind:=(riopnd← (riopnd + reg[set@x2] )<31:0>) )NEXT         !indexing
        loc ← (mar+2)<19:0> ;           !update loc
        instr←0                         !END OF INSTRUCTION MODE
END ;   !ri1format

                !ri type 2 instruction format

ri2format := begin
        mar← (mar+2)<19:0>; nbytes←2 NEXT
        memrd   NEXT                    !read 2 bytes more of instruction
        ir<16:31>←mbr<16:31> NEXT
        mar←(mar+2)<19:0> Next          !update mar to read last 2 bytes
        memrd Next                      !now have read 48-bits of instr
        ir<32:47>←mbr<16:31> Next                       !had to read 2 halfwords to
                        !insure proper boundary alignment
        loc ← (mar+2)<19:0> ; instr ←0 next             !update loc, instr
        riopnd←ri2 NEXT                                 !calculate ri operand
        (if x2 => ri2ind:=(riopnd← (riopnd + reg[set@x2])<31:0>) )      !indexing
END ;   !ri2format


                !rx format instructions

rxformat:=begin
        mar← (mar+2)<19:0>  NEXT                        !nbyte still 2
        memrd       NEXT                                !read another half word
        ir<16:31>←mbr<16:31> NEXT
        temp32←0   NEXT                                 !init
        (decode rxtype =>                               !determine rxtype
rx1format\00:=  begin                                   !rx type 1
                temp32←d2 NEXT                          !d2 nonneg
                (if x2 => rx1ind:=(temp32←(temp32 + reg[set@x2])<31:0> )) Next
                ea←temp32<12:31>                        !effective address
                END ;   !rx1

rx3format\01:=  begin                   !rx type 3
                mar←(mar+2)<19:0> NEXT
                memrd     NEXT                  !read another half word of instr
                ir<32:47>←mbr<16:31> NEXT
                Temp32←a2  NEXT
                (if fx2 => temp32←(temp32 + reg[set@fx2])<31:0> )  NEXT
                (if sx2 => temp32←(temp32 + reg[set@sx2])<31:0> )  next
                ea←temp32<12:31>;               !rx3 effective address
```

```
                (DECODE (ix2 NEQ 0) @ (sx2 NEQ 0) =>
                        \none   rx3none:=(nop);
                        \s      rx3s:=(nop);
                        \f      rx3f:=(nop);
                        \fs     rx3fs:=(nop)
                )               !this is for R-M measures only--ignore
                END ;   !rx3

rx2aformat\18:= begin                           !rx type 2 & d2 nonneg
                temp32←(mar+2+d2)<31:0> NEXT
                (if x2 => rx2aind:=(temp32←(temp32+reg[set@x2])<31:0> )) NEXT
                ea←temp32<12:31>                !20-bit address
                END ;           !rx2 & d2 nonneg

rx2bformat\11:= begin           !rx type 2 & d2 neg
                temp32← ((#777777 @d2 ) + mar + 2)<31:0>  NEXT  !expand d2 to 32-bits
                (if x2 => rx2bind:=(temp32←(temp32+reg[set@x2])<31:0> )) NEXT
                ea←(temp32 )<12:31>             !20-bit address
                END     !rx2 and d2 neg
        ) NEXT          !END decode rxtype
        !oc←(mar + 2)<19:0>; !nstr←0
END ;   !rxformat
```

```
ILLINST :=Begin          !!!legal instruction encountered
        NEWPSW←"30 NEXT !ADDRESS OF NEW PSW FOR ILL.INST.INT.HANDLER
        SYSINT           !SWAP PSWS
END ;   !ILLINST
!
```

```
INTERDATA 8/32 ISP DESCRIPTION

        ! THE LOAD INSTRUCTIONS

LR:=begin       !load reg
        RRFORMAT NEXT
        CCOP ← REG[SET@R2]NEXT
        REG[SET@R1] ← CCOP NEXT
        CCFIXED !SET CC
END ; !LR


LIS:=BEGIN      !LOAD IMMED.SHORT
        SFFORMAT  NEXT
        CCOP ← N NEXT
        REG[SET@R1] ← CCOP  NEXT
        CCFIXED
END ;     !LIS


LCS:=BEGIN      !LOAD COMPLEMENT SHORT
        SFFORMAT  NEXT
        CCOP←(MINUS("0000000 @N)<31:0>)<31:0> NEXT
        REG[SET@R1] ← CCOP  NEXT
        CCFIXED
END;     !LCS


Linst:=BEGIN    !LOAD
        RXFORMAT  NEXT  !CALCULATE EA
        NBYTES← 4;MAR←EA NEXT
        MEMRD  NEXT     !READ WORD FROM MEM
        CCOP←MBR  NEXT  !LOAD
        REG[SET@R1] ← CCOP  NEXT
        CCFIXED
END;    !LINST


LI:=BEGIN               !LOAD IMMED
        RI2FORMAT  NEXT
        CCOP← RIOPND Next       !COND CODE PARAM
        REG[SET@R1]←CCOP NEXT   !LOAD
        CCFIXED
end; !LI

LH:=BEGIN               !LOAD HALFWORD
        RXFORMAT  NEXT
        MAR←EA; NBYTES←2  NEXT
        MEMRD   NEXT
        (DECODE MBR<16>=>       !SIGN BIT
        \0.NONNEG BEGIN
                CCOP← MBR<16:31> Next
                REG[SET@R1]←CCOP
                end ;   !\0-DECODE
        \1.NEG BEGIN
                CCOP← ("FFFF @ MBR<16:31>)<31:0> Next
                REG[SET@R1]←CCOP
                END     !\1-DECODE
        ) NEXT  !end DECODE
        CCFIXED
end;  !LH


LHI:=Begin      !load halfword immediate
        riiformat Next
        ccop←riopnd  Next      . !riopnd already sign extended
        reg[set@r1]←ccop Next
        ccfixed
END;    !!hi


LA:=BEGIN       !LOAD ADDRESS
        RXFORMAT  NEXT
        REG[SET@R1]←EA  !20-BIT ADDRESS
end;  !LA
```

```
LHL:=BEGIN      !LOAD HALFWORD LOGICAL
        RXFORMAT  NEXT
        NBYTES←2; MAR←EA  NEXT
        MEMRD NEXT              !READ HALFWORD
        CCOP←MBR<16:31> NEXT    !ZERO-FILL
        REG[SET@R1]←MBR<16:31> NEXT
        CCFIXED
end ;   !LHL


LM:=BEGIN       !LOAD MULTIPLE
        RXFORMAT  NEXT
        NBYTES←4; MAR←EA; TEMP4←R1 NEXT !TEMP4 IS USED FOR LOOP COUNTER
LMULT:=(        MEMRD  NEXT
                REG[SET@TEMP4]←MBR  NEXT
                MAR←(MAR+4)<19:0> Next
                (IF temp4 LSS 15 => temp4←(temp4+1)<3:0> Next
                                    LMULT)
        ) !end LMULT
end ;  ! LM


LB:=BEGIN       !LOAD BYTE
        RXFORMAT    NEXT
        MAR←EA;  NBYTES←1    NEXT
        MEMRD    NEXT
        REG[SET@R1]← MBR<24:31>         !FORCE HIGH BITS TO ZERO
END;    !lb

LBR:=BEGIN      !LOAD BYTE REG
        RRFORMAT  Next
        REG[SET@R1]←REG[SET@R2]<24:31>
        !FORCE    HIGHBITS TO ZERO
END;    !LBR

STH:= BEGIN     !STORE HALFWORD
        RXFORMAT    NEXT
        MAR←EA; NBYTES←2;
        MBR<16:31>←REG[SET@R1]<16:31>    NEXT
        MEMWT
END;    ! STH


STB:=BEGIN      !STORE BYTE
        RXFORMAT        Next
        MAR←EA; NBYTES← 1 ;
        MBR<24:31>←REG[SET@R1]<24:31>    NEXT
        MEMWT
END;    !STB

STBR:=BEGIN     !STORE BYTE REGISTER
        RRFORMAT    NEXT
        REG[SET@R2]<24:31>←REG[SET@R1]<24:31>
END;    !STBR

STM:=BEGIN      !STORE MULTIPLE
        RXFORMAT    NEXT
        NBYTES←4;MAR←EA;TEMP4←R1  NEXT
        SMULT:=( MBR←REG[SET@TEMP4]    NEXT
                MEMWT    NEXT
                (IF temp4 LSS 15 =>
                MAR← (MAR+4)<19:0> ;TEMP4← (TEMP4+1)<3:0>    NEXT
                SMULT )
                )       !END SMULT
END;    !STM

ST :=BEGIN      !STORE
        RXFORMAT   NEXT
        MAR←EA; NBYTES←4 Next
        MBR←REG[SET@R1)  NEXT
        MEMWT
END;    !ST
```

INTERDATA 8/32 ISP DESCRIPTION

```
EXBR:=Begin                !exhange byte  reg
        rrformat Next
        reg[setar2]<16:31>← reg[setar1]<24:31> @ reg[setar1]<16:23>
End;    !exbr
```

```
INTERDATA 8/32 ISP DESCRIPTION

        !BOOLEAN INSTRUCTIONS

ORINST:= BEGIN          !OR REGISTER
        RRFORMAT NEXT
        ccop←reg[ser@r1] or reg[ser@r2] Next
        reg[ser@r1]←ccop  Next
        ccfixed         !set condition code
end;    !orinst


O:= begin               !or instr
        rxformat  Next
        nbytes←4; mar←ea Next   !prepare to fetch opnd
        memrd  Next             !read a word
        ccop← reg[ser@r1] or mbr<0:31>  Next
        ccfixed
end; !O


OI:= begin              !or immediate
        ri2format Next
        ccop←reg[ser@r1] or riopnd Next
        reg[ser@r1]← ccop Next
        ccfixed
end; !OI


OH:=Begin               !or halfword
        rxformat Next
        mar←ea ; nbytes←2 Next  !prepare to fetch data
        memrd Next              !fetch halfword data
        (IFF (mbr<16> EQL 0) THEN temp32←mbr<16:31>;  !test sign bit
        ELSE
        temp32← "ffff @ mbr<16:31>      !propagate sign bit(negative)
        ) Next  !end IFF
        ccop ← reg[ser@r1] OR temp32 Next
        reg[ser@r1]←ccop Next
        ccfixed
END;    !oh


OHI:=Begin              !or halfword immediate
        rilformat Next
        ccop←reg[ser@r1] OR riopnd Next !riopnd already sign extended
        reg[ser@r1]←ccop Next
        ccfixed
END;    !ohi


X:=Begin                        !exclusive or
        rxformat Next
        mar←ea; nbytes←4 Next   !prepare to fetch data
        memrd Next              !fetch fulword data
        ccop←reg[ser@r1] XOR mbr Next
        reg[ser@r1]←ccop Next
        ccfixed
END;            !x


XR:=Begin                       !exclusive or register
        rrformat Next
        ccop←reg[ser@r1] XOR reg[ser@r2] Next
        reg[ser@r1]←ccop Next
        ccfixed
END;    !xr


XI:=Begin                       !exclusive or immediate
        ri2format Next
        ccop←reg[ser@r1] XOR riopnd Next
        reg[ser@r1]←ccop Next
        ccfixed
END;    !xi
```

```
INTERDATA 8/32 ISP DESCRIPTION

XH:=Begin                          !exclusive or halfword
        rxformat Next
        mar←ea; nbytes←2 Next      !prepare to fetch data
        memrd Next                 !fetch fullword data
        (IFF (mbr<16> EQL 0) THEN temp32←mbr<16:31>;  !test sign bit
        ELSE temp32←"ffff @ mbr<16:31>
        !sign extend data
        ) Next  !end IFF
        ccop←reg[set@r1] XOR mbr Next
        reg[set@r1]←ccop Next
        ccfixed
END;    !xh


XHI:=Begin                         !exclusive or halfword immediate
        rilformat Next
        ccop←reg[set@r1] XOR riopnd Next !sign already extend
        reg[set@r1]←ccop Next
        ccfixed
END;    !xhi

Ninst:=BEGIN            !AND
        RXFORMAT    NEXT
        MAR←EA;  NBYTES←4;
        MEMRD    NEXT   ! FETCH OPERAND
        CCOP←REG[SET@R1) AND MBR    NEXT
        REG[SET@R1]←CCOP    NEXT
        CCFIXED !SET COND CODE
END;    !Ninst

NR:=BEGIN       !AND REG
        RRFORMAT    NEXT
        CCOP←REG[SET@R1) AND REG[SET@R2)    NEXT
        REG[SET@R1]←CCOP NEXT
        CCFIXED
END;    !NR

NI:=BEGIN       !AND IMMED
        RI2FORMAT    NEXT
        CCOP←REG[SET@R1) AND RIOPND     NEXT    !32-BIT IMMED. OPRND
        REG[SET@R1]←CCOP    NEXT
        CCFIXED
END;    !NI

NH:=BEGIN       !AND HALFWORD
        RXFORMAT    NEXT
        MAR←EA;NBYTES←2    NEXT
        MEMRD    NEXT
        (DECODE MBR<16> =>      !CHK SIGN BIT OF HALFWORD
                \0.POS    TEMP32←MBR<16:31>;
                \1.NEG    TEMP32←"FFFF@(MBR<16:31>)
                )NEXT   !END DECODE
        CCOP←REG[SET@R1) AND TEMP32 NEXT
        REG[SET@R1]←CCOP    NEXT
        CCFIXED
END;    !NH

NHI:=BEGIN      !AND HALFWORD IMMED
        RI2FORMAT    NEXT
        CCOP←REG[SET@R1) AND RIOPND    NEXT
        REG[SET@R1) ← CCOP    NEXT
        CCFIXED
END;    !NHI
```

```
        !SHIFT INSTRUCTIONS

SLL:=Begin                          !shift left logical
        rilformat Next
        temp33← ('0@reg[ser1])<32:0> ↑SL0 (riopnd<27:31>) Next
        !use low-order 5-bits of riopnd to determine shift count
        !produce a 33-bit result which includes the carry in bit 0
        ccop←temp33<1:32> ;
        reg[ser1]←ccop Next    !32-bit result
        ccfixed Next
        c←temp33<0>      !set carry bit of condition code
END;    !sll


SLLS:=Begin                         !shift left logical short
        sfformat Next
        temp33←('0@reg[ser1])<32:0> ↑SL0 n  Next
        ccop←temp33<1:32> ;
        reg[ser1]←ccop Next    !32-bit result
        ccfixed Next
        c←temp33<0>              !set carry bit of condition code
END;    !slls


SLHL:=Begin                         !shift left halfword logical
        rilformat Next
        temp17← ('0@reg[ser1]<16:31>) ↑SL0 (riopnd<28:31>) Next
        !use low-order 4-bits of riopnd to determine shift count
        !shift the low-order 16-bits only; retain the last bit shifted
        ccop<0:15>←temp17<1:16>;         !pass ccfixed a 16-bit result
        reg[ser1]<16:31>←ccop<0:15> Next      !16-bit result
        ccfixed Next
        c←temp17<0>      !set carry bit of condition code
END;    !slhl


SLHLS:=Begin                        !shift left halfword logical short
        sfformat Next
        temp17← ('0@reg[ser1]<16:31>) ↑SL0 n  Next
        !shift the low-order 16 bits only; retain the last bit shifted
        ccop<0:15>←temp17<1:16>;         !pass ccfixed a 16-bit result
        reg[ser1]<16:31>←ccop<0:15> Next      !16-bit result
        ccfixed Next
        c←temp17<0>              !set carry bit of condition code
END;    !slhls


SRL:=Begin                          !shift right logical
        rilformat Next
        temp33<0:31>← reg[ser1] Next  !left-justify in 33-bit reg
                !in order to retain the last bit shifted out
        temp33←temp33 ↑SR0 (riopnd<27:31>) Next
        !use low-order 5-bits of riopnd to determine shift count
        !produce a 33-bit result which includes the carry in bit 32
        ccop←temp33<0:31> ;
        reg[ser1]←ccop Next    !32-bit result
        ccfixed Next
        c←temp33<32>     !set carry bit of condition code
END;    !srl


SRLS:=Begin                         !shift right logical short
        sfformat Next
        temp33<0:31>←reg[ser1] Next   !left-justify into 33-bit reg
                !in order to retain the last bit shifted out
        temp33←temp33 ↑SR0 n  Next
        ccop←temp33<0:31> ;
        reg[ser1]←ccop Next    !32-bit result
        ccfixed Next
        c←temp33<32>             !set carry bit of condition code
END;    !srls


SRHL:=Begin                         !shift right halfword logical
```

```
        rilformat Next
        temp17<0:15>← reg[seter1]<16:31> Next   !left-justify in 17-bit reg
                !in order to retain the last bit shifted out
        temp17←temp17 ↑SR0 (riopnd<28:31>) Next
        !use low-order 4-bits of riopnd to determine shift count
        !produce a 17-bit result which includes the carry in bit 16
        ccop<0:15>←temp17<0:15>;
        reg[seter1]<16:31>←ccop<0:15> Next      !16-bit result
        ccfixed Next
        c←temp17<16>    !set carry bit of condition code
END;    !srhl

SRHLS:=Begin                    !shift right halfword logical short
        sfformat Next
        temp17<0:15>←reg[seter1]<16:31> Next    !left-justify into 17-bit reg
                !in order to retain the last bit shifted out
        temp17←temp17 ↑SR0 n  Next
        ccop<0:15>←temp17<0:15> ;
        reg[seter1]<16:31>←ccop<0:15> Next      !16-bit result
        ccfixed Next
        c←temp17<16>            !set carry bit of condition code
END;    !srhls

RLL:=Begin              !rotate left logical
        rilformat Next
        ccop←reg[seter1] ↑RL riopnd<27:31> Next  !32-bits
        !rotate left by the amt specified in the low 5 bits of riopnd
        reg[seter1]←ccop Next   !result
        ccfixed
END;    !rll

RRL:=Begin                      !rotate right logical
        rilformat Next
        ccop←reg[seter1] ↑RR riopnd<27:31> Next
        !rotate right by the amt specified in the low 5 bits of riopnd
        reg[seter1]←ccop Next
        ccfixed
END;    !rrl

SLA:=Begin                      !shift left arithmetic
        rilformat Next
        sign←reg[seter1]<0> ; temp32←reg[seter1]<1:31> Next
        !save sign of reg[seter1]; prepare to shift remaining bits
        temp32← (temp32 ↑SL0 riopnd<27:31>)<31:0> Next
        !shift by amt specified by low 5 bits of riopns
        ccop← sign @ temp32<1:31> Next  !the sign bit remains unchanged
        reg[seter1]←ccop Next
        ccfixed  Next           !set condition code
        c←temp32<0>     !set carry bit of cc
END;    !sla

SLHA:=Begin                     !shift left halfword aritmetic
        rilformat Next
        sign←reg[seter1]<16>; temp16←reg[seter1]<17:31> Next
        !shift low 15 bits; retain sign bit
        temp16← (temp16 ↑SL0 riopnd<28:31>)<15:0> Next
        !use low-order 4-bits of riopnd to determine shift count
        ccop<0:15>← sign @ temp16<1:15> Next
        reg[seter1]<16:31>← ccop<0:15> Next
        ccfixed Next
        c←temp16<0>     !set carry bit of cc
END;    !slha

SRA:=Begin                      !shift right arithmetic
        rilformat Next
        sign←reg[seter1]<0> ;   !retain sign
        temp32<0:30>←reg[seter1]<1:31> Next     ! do right 31-bit shift
        (DECODE sign =>         !sign determines fill bit
```

```
        \0.pos  temp32← temp32 ↑SR0 riopnd<27:31>;
        \1.neg  temp32← temp32 ↑SR1 riopnd<27:31>
        ) Next  !end decode
        !shifted right, sign fill; carry in bit 31 of temp32
        ccop← sign @ temp32<0:30> Next  !sign @ 31-bit result
        reg[set@rl]←ccop Next
        ccfixed Next
        c←temp32<31>                    !set carry bit of cc
END;    !sra


SRHA:=Begin                             !shift right halfword reg
        rilformat Next
        temp16<0:14>←reg[set@rl]<17:31>;        !15-bit shift to be done
        sign←reg[set@rl]<0> Next        !save sign
        (DECODE sign =>                 !sign determines fill bit
        \0.pos  temp16← temp16 ↑SR0 riopnd<28:31>;
        \1.neg  temp16← temp16 ↑SR1 riopnd<28:31>
        ) Next
        !shift right by amt specified in low 4 bits of riopnd
        !when shifting, propagate sign, and save carry in bit 15 of temp16
        ccop<0:15>← sign @ temp16<0:14> Next
        reg[set@rl]<16:31>←ccop<0:15> Next
        ccfixed Next
        c←temp16<15>                    !set carry in cc
END;    !srha

TS:=Begin                               !test and set
        rxformat Next
        mar←ea; nbytes←2 Next   !prepare to fetch halfword data
        memrd Next                      !fetch halfword
        l←mbr<16> Next                  !use most significant bit to set cc
        !NOTE: could have done ccop<0:15>←mbr<16:31>, and called ccfixed
        mbr←mbr OR "8000 Next   !set most significant bit of halfword
        memwt                           !write back halfword
END;    !ts


TLATE:=Begin            !translate
        rxformat Next
        mar←(ea + (('0@reg[set@rl]<24:31>)<8:0> ↑SL0 1))<19:0>;nbytes←2 Next
        !use character in reg to index into table at address ea
        memrd Next      !read halfword table entry into low half of mbr
        (IFF (mbr<16> EQL 1) THEN               !test most significant bit of table entry
                reg[set@rl]<24:31>←mbr<24:31>;
        !table contains a new transalted character
                ELSE
        tlatel:=(loc← ('0@mbr<17:31>)<15:0> ↑SL0 1)     !branch to translation routine
                !specified by table entry
        )       !end IFF
END;    !tlate
```

```
        !COMPARE INSTRUCTIONS

CInst:=Begin                    !compare
        rxformat Next
        mar←ea; nbytes←4 Next   !prepare to fetch data
        memrd Next                                          !fetch fullword operand
        temp33← (reg[seter1] + (NOT mbr) + 1)<32:0> Next
                        !subtract operands and compare result with zero
                        !perform "2**n + a - b" to cover all possible values of a & b
                        !(eg. b=(- 2**n) has no positive value, given n bits)
        ccop←temp33<1:32> Next                  !32-bit result
        ccfixed Next                            !set initial cond. code
        (IF l => c←1)                           !was result negative?
                        !carry set in cc when relation is <
END;                    !cinst


CR:=Begin                                       !compare reg.
        rrformat Next
        temp33← (reg[seter1] + (NOT reg[seter2]) + 1)<32:0> Next
                        !perform "2**n + a - b" to cover all values of a & b
                        !subtract operands and compare difference with zero
        ccop←temp33<1:32> Next                  !32-bit result
        ccfixed Next
        (IF l => c←1)
                        !carry set in cc when relation is <
END;                    !cr


CI:=Begin                                       !compare immediate
        ri2format Next
        temp33← (reg[seter1] +(NOT riopnd) + 1)<32:0> Next
                        !compute difference of operands and compare with zero
        ccop←temp33<1:32> Next
        c← NOT temp33<0>                        !set carry bit (flipped)
END;                    !ci


CH:=Begin                                       !compare halfword
        rxformat Next
        mar←ea; nbytes←2 Next                   !prepare to fetch halfword
        memrd Next                              !fetch halfword data
                        !sign-extend and convert data to fullword operand and do comparison
        (DECODE mbr<16> =>                      !test sign bit of halfword data
        \0.pos  temp33←(reg[seter1] + (NOT ("FFFF@mbr<16:31>) ) + 1)<32:0> ;
        \1.neg  temp33←(reg[seter1] +(NOT ("0000 @ mbr<16:31>) ) + 1)<32:0>
        ) Next                  !end decode
        ccop←temp33<1:32> Next
        ccfixed Next
        c← NOT (temp33<0>)                      !carry set for < relations
END;                    !ch


CHI:=Begin                                      !compare halfword immediate
        rilformat Next                          !riopnd already sign extended
        temp33←(reg[seter1] +(NOT riopnd) + 1)<32:0> Next
        ccop←temp33<1:32> Next
        ccfixed Next
        c← NOT temp33<0>        !set carry when relation is <
END;    !chi


CL:=Begin                       !compare logical
        rxformat Next
        mar←ea; nbytes←4 Next   !prepare to fetch data word
        memrd Next
        temp33←(reg[seter1] + (NOT mbr) + 1)<32:0> Next
        ccop←temp33<1:32> Next
        ccfixed Next            !compute arithmetic cc
        (DECODE reg[seter1]<0> @ mbr<0> =>      !signs of operands
                \00     (IF l => c←1);  !both positive;relation same as arithmetic compare
                \01     c←1;    !relation is <
                \10     nop;            !relation is >
                \11     (IF g => c←1)   !reverse relation
```

```
        )       !end decode
END;    !cl

CLR:=Begin                      !compare logical reg
        rrformat Next
        temp33←(reg[ster1] + (NOT reg[ster2]) + 1)<32:0> Next
        ccop←temp33<1:32> Next
        ccfixed Next            !compute arithmetic cc
        (DECODE reg[ster1]<0> @ reg[ster2]<0> =>        !signs of operands
                \00     (IF l => c←1);  !both positive;relation same as arithmetic compare
                \01     c←1;    !relation is <
                \10     nop;            !relation is >
                \11     (IF g => c←1)   !reverse relation
        )       !end decode
END;    !clr

CLI:=Begin                      !compare logical immed
        ri2format Next
        temp33←(reg[ster1] + (NOT riopnd) + 1)<32:0> Next
        ccop←temp33<1:32> Next
        ccfixed Next            !compute arithmetic cc
        (DECODE reg[ster1]<0> @ riopnd<0> =>    !signs of operands
                \00     (IF l => c←1);  !both positive;relation same as arithmetic compare
                \01     c←1;    !relation is <
                \10     nop;            !relation is >
                \11     (IF g => c←1)   !reverse relation
        )       !end decode
END;    !cli

CLH:=Begin                      !compare logical halfword
        rxformat Next
        mar←ea; nbytes←2 Next   !prepare to fetch data halfword
        memrd Next
        (IFF mbr<16> THEN temp32←mbr<16:31>;
                ELSE temp32← "ffff @ mbr<16:31>
        ) Next  !halfword data sign extended
        temp33←(reg[ster1] + (NOT temp32) + 1)<32:0> Next
        ccop←temp33<1:32> Next
        ccfixed Next            !compute arithmetic cc
        (DECODE reg[ster1]<0> @ temp32<0> =>    !signs of operands
                \00     (IF l => c←1);  !both positive;relation same as arithmetic compare
                \01     c←1;    !relation is <
                \10     nop;            !relation is >
                \11     (IF g => c←1)   !reverse relation
        )       !end decode
END;    !clh

CLHI:=Begin                     !compare logical halfword immed
        rilformat Next
        !riopnd already sign extended
        temp33←(reg[ster1] + (NOT riopnd) + 1)<32:0> Next
        ccop←temp33<1:32> Next
        ccfixed Next            !compute arithmetic cc
        (DECODE reg[ster1]<0> @ riopnd<0> =>    !signs of operands
                \00     (IF l => c←1);  !both positive;relation same as arithmetic compare
                \01     c←1;    !relation is <
                \10     nop;            !relation is >
                \11     (IF g => c←1)   !reverse relation
        )       !end decode
END;    !clhi

CLB:=Begin                      !compare logical byte
        rxformat Next
        mar←ea; nbytes←1 Next           !prpare to fetch byte data
        memrd Next
        temp8←(reg[ster1]<24:31> + (NOT mbr<24:31>) + 1 )<7:0> Next
        ccop<0:7>←temp8 Next
        ccfixed Next
        (DECODE reg[ster1]<24> @ mbr<24> =>     !signs of data
```

```
                \00     (IF l => c←1);  !both positive;relation same as arithmetic compare
                \01     c←1;    !relation is <
                \10     nop;            !relation is >
                \11     (IF g => c←1)   !reverse relation
        )       !end decode
END;    !clb


CHVR:=Begin                     !convert to halfword value reg
        rrformat Next
        temp32<16:31>←reg[ser2]<16:31> Next
        (DECODE temp32<16> =>   !halfword sign bit
        \0.pos  temp32<0:15>←0;
        \1.neg  temp32<0:15>←"ffff
        ) Next  !sign extended
        temp1←c;                !save current carry in cc
        ccop←temp32 ; reg[ser1]←temp32 Next
        ccfixed Next
        c←temp1 Next    !restore old carry
        (DECODE temp32<0> =>    !sign of result
        \0.pos  (IF reg[ser2]<0:15> NEQ 0 => v←1);
        \1.neg  (IF reg[ser2]<0:15> NEQ "ffff => v←1)
        )       !overflow occurred--- not a valid halfword
END;    !chvr
```

```
INTERDATA 8/32 ISP DESCRIPTION

        !BIT INSTRUCTIONS


TBT:=Begin                          !test bit
        rxformat Next
        mar←(ea + reg[set@rl] ↑SR0 3)<19:0> ; nbytes←1 Next
        !value in reg is a bit displacement into the array
        !starting at address ea.
        memrd Next                  !fetch byte from array
        (DECODE reg[set@rl]<29:31> =>   !get indicated bit
        \0      templ←mbr<24>;
        \1      templ←mbr<25>;
        \2      templ←mbr<26>;
        \3      templ←mbr<27>;
        \4      templ←mbr<28>;
        \5      templ←mbr<29>;
        \6      templ←mbr<30>;
        \7      templ←mbr<31>
        ) Next  !end bselect
        ccop←templ Next             !set cc's g if bit is 1
        ccfixed
END;    !tbt


CBT:=Begin                          !complement(flip) bit
        rxformat Next
        mar←(ea + reg[set@rl] ↑SR0 3)<19:0>;
        nbytes←1 Next               !fetch byte within bit array
        Memrd Next
        (DECODE reg[set@rl]<29:31> =>   !select bit within selected byte
        \0      (templ←mbr<24> Next
                        mbr<24>← NOT templ );
        \1      (templ←mbr<25> Next
                        mbr<25>← NOT templ );
        \2      (templ←mbr<26> Next
                        mbr<26>← NOT templ );
        \3      (templ←mbr<27> Next
                        mbr<27>← NOT templ );
        \4      (templ←mbr<28> Next
                        mbr<28>← NOT templ );
        \5      (templ←mbr<29> Next
                        mbr<29>← NOT templ );
        \6      (templ←mbr<30> Next
                        mbr<30>← NOT templ );
        \7      (templ←mbr<31> Next
                        mbr<31>← NOT templ )
        ) Next  !end decode bit select
        memwt Next                  !write back byte with bit flipped
        ccop←templ Next
        ccfixed         !set cc's g if bit was 1
END;    !cbt


SBT:=Begin                          !set bit
        rxformat Next
        mar←(ea + reg[set@rl] ↑SR0 3)<19:0>;
        nbytes←1 Next               !fetch byte within bit array
        Memrd Next
        (DECODE reg[set@rl]<29:31> =>   !select bit within selected byte
        \0      (templ←mbr<24> Next
                        mbr<24>← 1 );
        \1      (templ←mbr<25> Next
                        mbr<25>← 1 );
        \2      (templ←mbr<26> Next
                        mbr<26>← 1 );
        \3      (templ←mbr<27> Next
                        mbr<27>← 1 );
        \4      (templ←mbr<28> Next
                        mbr<28>← 1 );
        \5      (templ←mbr<29> Next
                        mbr<29>← 1 );
        \6      (templ←mbr<30> Next
                        mbr<30>← 1 );
        \7      (templ←mbr<31> Next
```

```
                        mbr<31>← 1 )
        ) Next  !end decode bit select
        memwt Next                  !write back byte with bit flipped
        ccop←tempi Next
        ccfixed             !set cc's g if bit was 1
END;    !sbt


RBT:=Begin                          !reset(clear) bit
        rxformat Next
        mar←(ea + reg[seter1] †SR0 3)<19:0>;
        nbytes←1 Next               !fetch byte within bit array
        Memrd Next
        (DECODE reg[seter1]<29:31> =>   !select bit within selected byte
        \0      (tempi←mbr<24> Next
                        mbr<24>←0 );
        \1      (tempi←mbr<25> Next
                        mbr<25>←0 );
        \2      (tempi←mbr<26> Next
                        mbr<26>←0 );
        \3      (tempi←mbr<27> Next
                        mbr<27>←0 );
        \4      (tempi←mbr<28> Next
                        mbr<28>←0 );
        \5      (tempi←mbr<29> Next
                        mbr<29>←0 );
        \6      (tempi←mbr<30> Next
                        mbr<30>←0 );
        \7      (tempi←mbr<31> Next
                        mbr<31>←0 )
        ) Next  !end decode bit select
        memwt Next                  !write back byte with bit flipped
        ccop←tempi Next
        ccfixed             !set cc's g if bit was 1
END;    !rbt
```

```
INTERDATA 8/32 ISP DESCRIPTION

        !ARITHMETIC INSTRUCTIONS

Ainst:=Begin            !ADD
        rxformat Next
        mar←ea ; nbytes←4 Next
        memrd Next              !fetch the operand (fullword)
        temp33 ← reg[ser@r1] + mbr Next !33-bit result
        ccop ← temp33<1:32> Next        !32-bit result
        ccfixed Next                    !set  cond code
        (if temp33<0> => c←1 ) Next     !test carry
        (if (mbr<0> EQL reg[ser@r1]<0>) AND (temp33<0> NEQ temp33<1>)
                =>      v←1
        ) Next          !test for overflow
        reg[ser@r1] ← ccop   !32-bit result
END ;  !Ainst


AR:=Begin                                       !add register
        rrformat Next
        temp33← reg[ser@r1] + reg[ser@r2] Next  !33-bit result
        ccop←temp33<1:32> Next                  !32-bit result
        ccfixed Next                            !set initial cc
        (IF temp33<0> => c←1) Next              !test for carry
        (IF (reg[ser@r1]<0> EQL reg[ser@r2]<0>) AND (temp33<0> NEQ temp33<1>)
                =>  v←1
        ) Next                                  !test for overflow
        reg[ser@r1]←ccop                        !32-bit result
END;    !ar


AI:=Begin                                       !add immediate
        ri2format Next
        temp33← reg[ser@r1] + riopnd Next       !33-bit result
        ccop←temp33<1:32> Next                  !32-bit result
        ccfixed Next                            !set initial cc
        (IF temp33<0> => c←1) Next              !test for carry
        (IF (reg[ser@r1]<0> EQL riopnd<0>) AND (temp33<0> NEQ temp33<1>)
                =>  v←1
        ) Next                                  !test for overflow
                                                !if the signs of the two operands are the same
                                                !and these differ from that of the result
                                                !then overflow occurred
        reg[ser@r1]←ccop                        !32-bit result
END;    !ai


AIS:=Begin                                                      !add immed. short
        sfformat Next
        temp32←n Next                           !expand to 32 bits (zero-fill)
        temp33← reg[ser@r1] + temp32 Next       !33-bit result
        ccop←temp33<1:32> Next                  !32-bit result
        ccfixed Next                            !set initial cc
        (IF temp33<0> => c←1) Next              !test for carry
        (IF (reg[ser@r1]<0> EQL temp32<0>) AND (temp33<0> NEQ temp33<1>)
                =>  v←1
        ) Next                                  !test for overflow
        reg[ser@r1]←ccop                        !32-bit result
END;    !ais


AH:=Begin                                       !add halfword
        rxformat Next
        mar←ea; nbytes←2 Next                   !prepare to fetch halfword data
        memrd Next
        (IF mbr<16> => mbr<0:15>←"ffff) Next    !sign extend halfword data
        temp33←reg[ser@r1] + mbr Next           !33-bit result
        ccop←temp33<1:32> Next                  !32-bit result
        ccfixed Next                            !set initial cc
        (IF temp33<0> => c←1) Next              !test for carry
        (IF (reg[ser@r1]<0> EQL mbr<0>) AND (temp33<0> NEQ temp33<1>)
                =>  v←1
        ) Next                                  !test for overflow
        reg[ser@r1]←ccop                        !32-bit result
END;    !ah
```

```
AHI:=Begin                                          !add halfword immediate
        riiformat Next
                                                    !riopnd already sign extended
        temp33 ← reg[ser1] + riopnd Next            !33-bit result
        ccop ← temp33<1:32> Next                    !32-bit result
        ccfixed Next                                !set  cond code
        (if temp33<0> => c←1 ) Next                 !test carry
        (if (reg[ser1]<0> EQL riopnd<0>) AND (temp33<0> NEQ temp33<1>)
                =>      v←1
        ) Next                                      !test for overflow
        reg[ser1] ← ccop                            !32-bit result
END ;   !AHI


AM:=Begin                                           !add to memory
        rxformat Next
        mar←ea ; nbytes←4 Next
        memrd Next              !fetch the operand (fullword)
        temp33 ← reg[ser1] + mbr Next !33-bit result
        ccop ← temp33<1:32> Next        !32-bit result
        ccfixed Next                    !set  cond code
        (if temp33<0> => c←1 ) Next     !test carry
        (if (mbr<0> EQL reg[ser1]<0>) AND (temp33<0> NEQ temp33<1>)
                =>      v←1
        ) Next          !test for overflow
        mbr←ccop Next                   !prepare to store result back in memory
        Memwt                           !write fullword result at address ea
End;    !am


AHM:=Begin                                          !add halfword memory
        rxformat Next
        mar←ea; nbytes←2 Next                       !prepare to fetch halfword data
        memrd Next
        (IF mbr<16> => mbr<0:15>←"ffff) Next        !sign extend halfword data
        temp17← (reg[ser1] + mbr)<16:0>  Next !17-bit result
        ccop<0:15>←temp17<1:16> Next                !16-bit result
        ccfixed Next                                !set initial cc
        (IF temp17<0> => c←1) Next                  !test for carry
        (IF (reg[ser1]<0> EQL mbr<16>) AND (temp17<0> NEQ temp17<1>)
                =>  v←1
        ) Next                                      !test for overflow based on halfword result
        mbr←temp17<1:16> Next                       !16-bit result
        Memwt                                       !write halfword result back to memory at address ea
END;    !ahm
S:=Begin                                            !subtract
        rxformat Next
        mar←ea ; nbytes←4 Next
        memrd Next                                  !fetch the operand (fullword)
        temp33 ← (reg[ser1] + (NOT mbr) + 1)<32:0> Next       !33-bit result
        ccop ← temp33<1:32> Next                    !32-bit result
        ccfixed Next                                !set  cond code
        (if temp33<0> => c←1 ) Next                 !test borrow
        (if (mbr<0> NEQ reg[ser1]<0>) AND (temp33<0> NEQ temp33<1>)
                =>      v←1
        ) Next                                      !test for overflow
        reg[ser1] ← ccop                            !32-bit result
END ;   !s


SR:=Begin                                           !substract register
        rrformat Next
        temp33← (reg[ser1] + (NOT reg[ser2]) + 1)<32:0> Next        !33-bit result
        ccop←temp33<1:32> Next                      !32-bit result
        ccfixed Next                                !set initial cc
        (IF temp33<0> => c←1) Next                  !test for borrow
        (IF (reg[ser1]<0> NEQ reg[ser2]<0>) AND (temp33<0> NEQ temp33<1>)
                =>  v←1
        ) Next                                      !test for overflow
        reg[ser1]←ccop                              !32-bit result
END;    !sr
```

```
SI:=Begin                                       !subtract immediate
        ri2format Next
        temp33← (reg[set@r1] + (NOT riopnd) + 1)<32:0> Next     !33-bit result
        ccop←temp33<1:32> Next                  !32-bit result
        ccfixed Next                            !set initial cc
        (IF temp33<0> => c←1) Next              !test for borrow
        (IF (reg[set@r1]<0> NEQ riopnd<0>) AND (temp33<0> NEQ temp33<1>)
                => v←1
        ) Next                                  !test for overflow
                                                !if the signs of the two operands are the same
                                                !and these differ from that of the result
                                                !then overflow occurred
        reg[set@r1]←ccop                        !32-bit result
END;    !si


SIS:=Begin                                                      !subtract immed. short
        sfformat Next
        temp32←n Next                           !expand to 32 bits (zero-fill)
        temp33← (reg[set@r1] + (NOT temp32) + 1)<32:0> Next     !33-bit result
        ccop←temp33<1:32> Next                  !32-bit result
        ccfixed Next                            !set initial cc
        (IF temp33<0> => c←1) Next              !test for borrow
        (IF (reg[set@r1]<0> NEQ temp32<0>) AND (temp33<0> NEQ temp33<1>)
                => v←1
        ) Next                                  !test for overflow
        reg[set@r1]←ccop                        !32-bit result
END;    !sis


SH:=Begin                                       !subtract halfword
        rxformat Next
        mar←ea; nbytes←2 Next                   !prepare to fetch halfword data
        memrd Next
        (IF mbr<16> => mbr<0:15>←"ffff) Next    !sign extend halfword data
        temp33← (reg[set@r1] + (NOT mbr) + 1)<32:0> Next                !33-bit result
        ccop←temp33<1:32> Next                  !32-bit result
        ccfixed Next                            !set initial cc
        (IF temp33<0> => c←1) Next              !test for borrow
        (IF (reg[set@r1]<0> NEQ mbr<0>) AND (temp33<0> NEQ temp33<1>)
                => v←1
        ) Next                                  !test for overflow
        reg[set@r1]←ccop                        !32-bit result
END;    !sh

SHI:=Begin                                      !subtract halfword immediate
        rilformat Next
                                                !riopnd already sign extended
        temp33 ← (reg[set@r1] + (NOT riopnd) + 1)<32:0> Next    !33-bit result
        ccop ← temp33<1:32> Next                !32-bit result
        ccfixed Next                            !set  cond code
        (if temp33<0> => c←1 ) Next             !test borrow
        (if (reg[set@r1]<0> NEQ riopnd<0>) AND (temp33<0> NEQ temp33<1>)
                =>      v←1
        ) Next                                  !test for overflow
        reg[set@r1] ← ccop                      !32-bit result
END ;   !SHI


Minst:=Begin                                    !multiply instruction
        rxformat Next
        !instruction requires an even/odd register pair
        !r1 must specify an EVEN register => r1 and (r1 +1) selected
        (IF r1<3> => nop) Next                  !if r1 not even =>error---garbage results!!!
        mar←ea; nbytes←4 Next                   !prepare to fetch data operand
        memrd Next                              !fetch fullword data
        sign←0 ;  temp32←reg[set@(r1 OR 1)] Next                !need to determine sign of result
                                                !both operands must be positive for a logical multiply
        (IF mbr<0> => mbr←(MINUS mbr)<31:0>; sign←(sign+1)<0>) Next
        (IF temp32<0> => temp32←(MINUS temp32)<31:0>; sign←(sign+1)<0>) Next
        temp64← temp32 * mbr Next               !compute 64-bit product
        (IF sign => temp64←(MINUS temp64)<63:0>) Next   !check sign of result
        reg[set@r1]←temp64<0:31>;               !place most significant part of result in even register of the
        reg[set@(r1 OR 1)]←temp64<32:63>        pair !place least significant part of result in odd register
```

```
End;    !Minst

MR:=Begin                                       !multiply register instruction
        rrformat Next
        !instruction requires an even/odd register pair
        !r1 must specify an EVEN register => r1 and (r1 +1) selected
        (IF r1<3> => nop) Next                  !if r1 not even =>error---garbage results!!!
        sign←0 ;  temp32←reg[set@r2) ;          !need to determine sign of result
        dat32←reg[set@(r1 OR 1)) Next                   !both operands must be positive for a logical multiply
        (IF dat32<0> => dat32←(MINUS dat32)<31:0>; sign←(sign+1)<0>) Next
        (IF temp32<0> => temp32←(MINUS temp32)<31:0>; sign←(sign+1)<0>) Next
        temp64← temp32 * dat32 Next             !compute 64-bit product
        (IF sign => temp64←(MINUS temp64)<63:0>) Next   !check sign of result
        reg[set@r1]←temp64<0:31>;               !place most significant part of result in even register of the
        reg[set@(r1 OR 1)]←temp64<32:63>        pair  !place least significant part of result in odd register
End;    !MR


MH:=Begin                                       !multiply halfword instruction
        rxformat Next
        mar←ea; nbytes←2 Next                   !prepare to fetch halfword data operand
        memrd Next                              !fetch halfword data
        sign←0; temp16←reg[set@r1]<16:31> Next  !a logical multiply requires positive operands
        (IF temp16<0> => sign←(sign+1)<0>; temp16←(MINUS temp16)<15:0>)Next
        (IF mbr<16> => sign←(sign+1)<0>; mbr←(MINUS mbr<16:31>)<15:0>)Next
        temp32← temp16 * mbr<16:31> Next        !compute 32-bit product
        (IF sign => temp32←(MINUS temp32)<31:0>) Next   !check sign of result
        reg[set@r1]←temp32                      !store result in reg
End;    !MH


MHR:=Begin                                      !multiply halfword register
        rrformat Next
        sign←0; temp16←reg[set@r1]<16:31>;
        dat16←reg[set@r2]<16:31> Next   !logical multiply requires positive operands
        (IF dat16<0> => sign←(sign+1)<0>; dat16←(MINUS dat16)<15:0>)Next
        (IF temp16<0> => sign←(sign+1)<0>; temp16←(MINUS temp16)<15:0>) Next
        temp32←temp16 * dat16 Next              !compute 32-bit product
        (IF sign => temp32←(MINUS temp32)<31:0>) Next   (IF sign => temp32←(MINUS temp32)<31:0>) Next   !check
        reg[set@r1]←temp32
End;    !MHR                                            sign


Dinst:=Begin                            !divide
        rxformat Next
                                                !an even/odd register pair is required
        (IF r1<3> => nop) Next                  !error r1 must  be EVEN---garbage results
        fixed.float←0 Next                      !fixed pt arithmetic operation
                                        !fixed.float is a parameter passed to the arithchk routine (if an
        mar←ea; nbytes←4 Next   arithmetic !fetch 32-bit divisor
        Memrd Next
        (IFF (mbr EQL 0) THEN arithchk;                 !cannot divide by zero
              ELSE Begin
        temp64←reg[set@r1] @ reg[set@(r1 OR 1)] Next    !64-bit dividend: most significant bits in even reg
        sign←0 Next                             !initialize sign of quotient
                                                !determine sign of quotient and force operands to be positive
        (IF temp64<0> => sign←(sign+1)<0>;
                         temp64← (MINUS temp64)<63:0>   !dividend must be positive for logical divide
        ) Next
        (IF mbr<0> => sign←(sign+1)<0>;
                         mbr← (MINUS mbr)<31:0>                 !divisor must be positive
        ) Next
        Q64← temp64/mbr Next                                    !calculate logical quotient
        ovfchk:=(IFF q64<0:32> THEN arithchk;                   !test for overflow
                                                !quotient must be positive 32-bits
                         ELSE Begin
                temp32← (temp64 + (NOT  (q64<32:63> * mbr)) + 1)<31:0> Next
                                                        !calculate remainder
                (DECODE sign =>                         !convert quotient and remainder to the
                \0.pos  Begin           appropriate     !result is positive
                        reg[set@r1]←temp32;
                        reg[set@(r1 OR 1)]←q64<32:63>
                        end;
                \1.neg  Begin                           !result is negative
```

```
                        reg[set@r1]← (MINUS temp32)<31:0>;
                        reg[set@(r1 OR 1)]←(MINUS q64<32:63>)<31:0>
                        end
                )               !end DECODE
                        End             !ovfl ELSE
        )       !end ovfchk
                End             !ELSE
        )       !end IFF
End;            !dinst


DR:=Begin                               !divide register
        rrformat Next
                                                !an even/odd register pair is required
        (IF r1<3> => nop) Next                  !error r1 must  be EVEN---garbage results
        fixed.float←0 Next                      !fixed pt arithmetic operation
                                        !fixed.float is a parameter passed to the arithchk routine (if an
        (IFF (reg[set@r2] EQL 0) THEN arithchk;arithmetic    !cannot divide by zero
                ELSE Begin
        temp64←reg[set@r1] @ reg[set@(r1 OR 1)] Next    !64-bit dividend: most significant bits in even reg
        sign←0 Next                             !initialize sign of quotient
                                                !determine sign of quotient and force operands to be positive
        (IF temp64<0> => sign←(sign+1)<0>;
                        temp64← (MINUS temp64)<63:0>    !dividend must be positive for logical divide
        ) Next
        div32←reg[set@r2] Next                  !the 32-bit divisor
        (IF div32<0> => sign←(sign+1)<0>;
                        div32← (MINUS div32)<31:0>                      !divisor must be positive
        ) Next
        Q64← temp64/div32 Next                                  !calculate logical quotient
        ovfchk:=(IFF q64<0:32> THEN arithchk;                   !test for overflow
                                                        !quotient must be positive 32-bits
                        ELSE Begin
                temp32← (temp64 + (NOT (q64<32:63> * div32)) + 1)<31:0> Next
                                                                !calculate remainder
                (DECODE sign =>                                 !convert quotient and remainder to the
                \0.pos  Begin                   appropriate     !result is positive
                        reg[set@r1]←temp32;
                        reg[set@(r1 OR 1)]←q64<32:63>
                        end;
                \1.neg  Begin                                   !result is negative
                        reg[set@r1]← (MINUS temp32)<31:0>;
                        reg[set@(r1 OR 1)]←(MINUS q64<32:63>)<31:0>
                        end
                )               !end DECODE
                        End             !ovfl ELSE
        )       !end ovfchk
                End             !ELSE
        )       !end IFF
End;            !dr
```

```
INTERDATA 8/32 ISP DESCRIPTION

        !BRANCH INSTRUCTIONS

TI:=BEGIN       !TEST IMMEDIATE
        RI2FORMAT    NEXT
        CCOP←REG[SET@R1) AND R1OPND NEXT
        CCFIXED
END;    !TI

THI:=BEGIN      !TEST HALFWORD IMMED
        RI1FORMAT    NEXT
        CCOP←REG[SET@R1]AND R1OPND    NEXT
        CCFIXED
END;    !THI

BFC:=BEGIN      !BRANCH ON FALSE COND
        RXFORMAT    NEXT
        (IF (CC AND R1) EQL 0 =>
        LOC←EA AND "FFFFE
        )       !MUST BE ON HALFWORD BOUNDARY
END;    !BFC

BFCR:=BEGIN     !BRANCH ON FALSE REG
        RRFORMAT    NEXT
        (IF (CC AND R1) EQL 0 =>
                LOC←(REG[SET@R2) AND "FFFFFFFE) <19:0>
        )       !HALFWORD BOUNDARY
END;    !

BFBS:=BEGIN     !BRANCH FALSE BACKWARDS SHORT
        SFFORMAT    NEXT
        (IF (CC AND R1) EQL 0 =>
        LOC←(LOC - ((N@0)+2)) <19:0>
        )       !BRANCH N HALFWORDS BACKWARDS
END;    !BFBS

BFFS:=BEGIN     !BRANCH FALSE FORWARD SHORT
        SFFORMAT    NEXT
        (IF (CC AND R1) EQL 0 =>        !NONE OF THE CONDITIONS TRUE
        LOC←(LOC+((N@0) - 2))<19:0>
        )       !BRANCH N HALFWORDS FORWARDS
END;    !BFFS

BTC:=BEGIN      !BRANCH ON TRUE COND)
        RXFORMAT    NEXT
        (IF (CC AND R1) NEQ 0 =>
        LOC←EA AND "FFFFE       !IF ANY COND TRUE => BRANCH
        ) !END IF       !MUST BE ON HALFWORD BNDRY
END;    !BTC

BTCR:=BEGIN     !BRANCH ON TRUE COND REG
        RRFORMAT    NEXT
        (IF (CC AND R1) NEQ 0 =>
           LOC ←(REG[SET@R2) AND "FFFFFFFE)<19:0>
        )       !MUST BE ON HALFWORD BNDRY
END;    !BTCR

BTBS:=BEGIN     !BRANCH ON TRUE BACKWARD SHORT
        SFFORMAT   NEXT
        (IF (CC AND R1) NEQ 0 =>
        LOC←(LOC - ((N@0)+2)) <19:0>
        )       !BRANCH N HALFWORDS BACKWARDS
END;    !BTBS

BTFS:=BEGIN     !BRANCH TRUE FORWARD SHORT
        SFFORMAT    NEXT
        (IF (CC AND R1) NEQ 0 =>
        LOC←(LOC+((N@0) - 2))<19:0>
        )       !BRANCH N HALFWORDS FORWARDS
END;    !BTFS


BAL:= Begin             !branch and link
        rxformat Next
        reg[set@r1] ← loc Next  !save address of next instruction
```

```
        loc ← ea AND "fffffe      !address must be on halfword bndry
END; !BAL


BALR:=Begin             !branch and link register
        rrformat Next
        reg[seter1]← loc  Next  !save current loc (next instr addr)
        loc ← (reg[seter2] AND "fffffffe )<19:0>        !branch
                        !address must be on halfword bndry
END;  !BALR


BXLE:=Begin             !branch on index low or equal
        rxformat Next
        reg[seter1]← (reg[seter1] + reg[set@((r1+1)<3:0>)])<31:0> Next
        (if reg[seter1] leq reg[set@((r1+2)<3:0>)] =>
                loc←ea AND "fffffe  )  !warning: branch addr must be on
                                       ! halfword bndry
end;    !BXLE


BXH:=Begin              !branch on index high
        rxformat Next
        reg[seter1]← (reg[seter1] + reg[set@((r1+1)<3:0>)])<31:0> Next
        (if reg[seter1] gtr reg[set@((r1+2)<3:0>)] =>
                loc←ea AND "fffffe )
        !warning: branch addr must be on halfword bndry
end;    !bxh
```

```
INTERDATA 8/32 ISP DESCRIPTION

        !CIRCULAR LIST INSTRUCTIONS

ATL:=Begin                                  !add to top of list
        rxformat Next                       !get address of list headr
        mar←ea; nbytes←4 Next               !prepare to fetch list headr
        Memrd  Next                         !read first word of list headr
        max16←mbr<0:15> Next                !first halfword contains the max. no. of slots in list
                                            !2nd halfword contains the no. of slots used
        (IFF (max16 LEQ mbr<16:31>) THEN cc←4;
                                            !list full => overflow
                ELSE Begin
        mar← (ea + 2)<19:0>; mbr← mbr<16:31> + 1 ; nbytes←2 Next
                                            !increment the no. of slots used
        memwt Next                          !and update list headr
        mar← (ea + 4)<19:0> Next            !prepare to read another halfword of the list headr
        memrd Next                          !get slot no. of current list top
        (IFF (mbr EQL 0) THEN mbr←max16;        !max. no. of slots in list
                                            !list "wrap-around"
                ELSE mbr←mbr - 1 ) Next
                                            !mbr now contains slot no. of  current list top
        Memwt Next                          !update ptr to current top of list in headr
        mar←(ea + 8 + (mbr †SL0 2))<19:0> Next
                                            !mar contains the address of the current top of the list
        mbr← reg[setθrl] Next               !the data to be added to list
        Memwt Next                          !add element to list
        cc←0                                !list updated successfully
                End     !else
        )       !end IFF
END;    !atl


ABL:=Begin                                  !add to bottom of list
        rxformat Next                       !get address of list headr
        mar←ea; nbytes←4 Next               !prepare to fetch list headr
        Memrd  Next                         !read first word of list headr
        max16←mbr<0:15> Next                !first halfword contains the max. no. of slots in list
                                            !2nd halfword contains the no. of slots used
        (IFF (max16 LEQ mbr<16:31>) THEN cc←4;
                                            !list full => overflow
                ELSE Begin
        mar← (ea + 2)<19:0>; mbr← mbr<16:31> + 1 ; nbytes←2 Next
                                            !increment the no. of slots used
        memwt Next                          !and update list headr
        mar← (ea + 6)<19:0> Next                !prepare to read another halfword of the list headr
        memrd Next                          !get slot no. of next list bottom
        temp16←mbr<16:31> Next                  !save slot no. of  next list bottom
        mar←(ea + 8 + (mbr †SL0 2))<19:0> Next
                                            !mar contains the address of the next bottom of the list
        mbr←reg[setθrl]; nbytes←4 Next      !the data to be added to list
        Memwt Next                          !add element to list
        temp16← (temp16 + 1)<15:0> Next     !update slot no. of next bottom of list
        (IF temp16 GTR max16 => temp16←0) Next
                                            !test for list "wrap-around"
        mbr←temp16;
        mar←(ea + 6)<19:0>; nbytes←2 Next       !prepare to update list headr
        memwt Next                          !update slot no. of next list bottom
        cc←0                                !list updated successfully
                End     !else
        )       !end IFF
END;    !abl


RTL:=Begin                                                  !remove from top of list
        rxformat Next
        mar←ea; nbytes←4 Next                               !prepare to read list headr
        Memrd Next                                          !read 1st word of list headr
        max16←mbr<0:15> Next                                !1st halfword is the max. no. of slots in the list
        (IFF (mbr<16:31> EQL 0) THEN cc←4;                  !test for underflow (ie. list empty)
                ELSE Begin
        mbr←mbr<16:31> - 1; nbytes←2 ;                      !decrement no. of slots used
        mar←(ea + 2)<19:0> Next
        Memwt Next                                          !update part of list headr
        (IFF mbr THEN cc←2; ELSE cc←0) Next                 !will list be empty set cc
        mar←(ea + 4)<19:0> Next
```

```
        memrd Next                                      !fetch slot no. of current top of list
        temp16←mbr<16:31> Next                          !current top of list slot no.
        mar←(ea + 8 + (("0@temp16) ↑SL0 2))<19:0>;              !calculate address of slot at top of list
        nbytes←4 Next
        memrd Next                                      !remove data from slot
        reg[set@rl]←mbr Next
        temp16←(temp16 + 1)<15:0> Next                  !increment slot no. of current top of list
        (IF temp16 GTR max16 => temp16←0)Next           !test for list "wrap-around"
        mbr←temp16; nbytes←2; mar← (ea + 4)<19:0> Next
        Memwt                                           !update current top of list in list headr
                End     !ELSE
        )       !end IFF
End;    !rtl


RBL:=Begin                                      !remove from bottom of list
        rxformat Next
        mar←ea; nbytes←4 Next                   !prepare to read list headr
        Memrd Next                              !read 1st word of list headr
        max16←mbr<0:15> Next                    !1st halfword of headr contains the max no. of slots in the list
        (IFF (mbr<16:31> EQL 0) THEN cc←4;      !test for underflow (ie. list empty)
                ELSE Begin
        mbr←mbr<16:31> - 1;
        mar←(ea + 2)<19:0>;  nbytes←2 Next      !decrement the no. of slots used
        Memwt Next
        (IFF mbr NEQ 0 THEN cc←2; ELSE cc←0) Next       !set cc according to new list status (empty vs  non-empt
        mar←(ea+6)<19:0> Next                   !fetch slot no. of next bottom of list
        Memrd Next
        (IFF mbr NEQ 0 THEN mbr← mbr<16:31> - 1;        !update next bottom of list
                ELSE mbr←max16                  !check for list "wrap-around"
        ) Next
        Memwt Next                              !update list headr next bottom
        mar←(ea+8+ (mbr ↑SL0 2))<19:0>;         !calculate the address of the elt. at the bottom of the list
        nbytes←4 Next
        memrd Next                              !remove elt from bottom of list
        reg[set@rl]←mbr                         !store list elt in reg
                End     !ELSE
        )       !end IFF
End;    !rbl
```

```
INTERDATA 8/32 ISP DESCRIPTION

        !PRIVILEDGED INSTRUCTIONS

LPSW:=BEGIN                         !load program status word
        (IFF P THEN ILLINST  ;      !PROTECT MODE ON
        ELSE BEGIN
        RXFORMAT NEXT    !COMPLETE INSTR FETCH AND CALCULATE EA
        (IF EA<17:19> NEQ 0=>       !MUST BE ON DOUBLE WORD BNDRY
                EA<17:19>←0  )  NEXT    !FORCE IT
        MAR←EA NEXT
        TEMP64<0:31>←WMEM[WMAR] NEXT
        MAR←(MAR+4)<19:0>  NEXT
        TEMP64<32:63>←WMEM[WMAR]  NEXT
        PSW←TEMP64 NEXT
  !PSW←WMEM[EA]@WMEM[EA+8]
        QSCHK    !CHECK Q SERVICE
        END ! ELSE
        )! IFF END
END ; !LPSW



EPSR:=BEGIN                         !exchange program status register
        (IFF P THEN ILLINST  ;      !PROTECT MODE ON
                 ELSE BEGIN         !XQT
                reformat NEXT
                REG[SETe"1]←PSW<0:31> NEXT
                PSW<0:31> ← REG[SETe"2] NEXT
                QSCHK
                END      !XQT
        )       ! IFF END
END ;   !EPSR


SVC:=BEGIN                          !supervisor call
        RXFORMAT  NEXT !
        OLDPSW← PSW NEXT            !SAVE CURRENT PSW
        MAR←"98 NEXT    !ADDR OF NEW STATUS
        PSW<0:31>←WMEM[WMAR]   NEXT     ! NEW STATUS
        MAR← ((^8@R1) ↑SL0 1) + "9C   NEXT      !ADDR OF NEW LOC
        LOC←MEM[MMAR] NEXT      ! NEW LOC
        REG[SETe"0]←EA ;            !PASS PARAMETER
        REG[SETe"E]←OLDPSW<0:31> ;      !PASS OLD PSW
        REG[SETe"F]←OLDPSW<32:63>
END ; ! SVC


LPSWR:=BEGIN                        !load program status word reg
        (IFF P THEN ILLINST ;       ! PROTECT MODE
                ELSE BEGIN      !XQT
        PSW ← REG[SETe"R2)@REG[SETe((R2+1)<3:0>)] NEXT
        QSCHK    !CHK QSERVICE
                END !ELSE
        )       !IFF END
END ;   ! LPSWR
```

```
        !UNIMPLEMENTED INSTRUCTIONS

DHR:=Begin
        rrformat
End;    !dhr

LER:=Begin
        rrformat
End;    !ler

CER:=Begin
        rrformat
End;    !cer

AER:=Begin
        rrformat
End;    !aer

SER:=Begin
        rrformat
End;    !ser

MER:=Begin
        rrformat
End;    !mer

DER:=Begin
        rrformat
End;    !der

FXR:=Begin
        rrformat
End;    !fxr

FLR:=Begin
        rrformat
End;    !flr

EXHR:=Begin
        rrformat
End;    !exhr

DH:=Begin
        rxformat
End;    !dh

STE:=Begin
        rxformat
End;    !ste

LE:=Begin
        rxformat
End;    !le

CE:=Begin
        rxformat
End;    !ce

AE:=Begin
        rxformat
End;    !ae

SE:=Begin
        rxformat
End;    !se

ME:=Begin
        rxformat
End;    !me

DE:=Begin
        rxformat
End;    !de
```

```
STME:=Begin
        rxformat
End;    !stme

LME:=Begin
        rxformat
End;    !lme

CRC12:=Begin
        rxformat
End;    !crc12

CRC16:=begin
        rxformat
End;    !crc16

WBR:=Begin
        rrformat
End;    !wbr

RBR:=Begin
        rrformat
End;    !rbr

WHR:=Begin
        rrformat
End;    !whr

RHR:=Begin
        rrformat
End;    !rhr

WDR:=Begin
        rrformat
End;    !wdr

RDR:=Begin
        rrformat
End;    !rdr

SSR:=Begin
        rrformat
End;    !ssr

OCR:=Begin
        rrformat
End;    !ocr

AL:=Begin
        rxformat
End;    !al

WB:=Begin
        rxformat
End;    !wb

RB:=Begin
        rxformat
End;    !rb

WH:=Begin
        rxformat
End;    !wh

RH:=Begin
        rxformat
End;    !rh

WD:=Begin
        rxformat
End;    !wd

RD:=Begin
```

```
        rxformat
End;    !rd

SS:=Begin
        rxformat
End;    !ss

OC:=Begin
        rxformat
End;    !oc

SINT:=Begin
        rilformat
End;    !sint

SCP:=Begin
        rxformat
End;    !scp


LDR:=Begin
        rrformat
End;    !ldr

CDR:=Begin
        rrformat
End;    !cdr


ADR:=Begin
        rrformat
End;    !adr

SDR:=begin
        rrformat
End;    !sdr

MDR:=Begin
        rrformat
End;    !mdr

DDR:=begin
        rrformat
End;    !ddr

FXDR:=Begin
        rrformat
End;    !fxdr

FLDR:=Begin
        rrformat
End;    !fldr

LRA:=Begin
        rxformat
End;

STD:=begin
        rxformat
End;

LD:=Begin
        rxformat
End;

cd:=Begin
        rxformat
end;

ad:=begin
        rxformat
end;

sd:=begin
```

```
        rxformat
end;

md:=begin
        rxformat
end;

dd:=begin
        rxformat
end;

stmd:=begin
        rxformat
end;

lmd:=begin
        rxformat
end;
```

```
INTERDATA 8/32 ISP DESCRIPTION

        !EMULATION ROUTINES

!
IFETCH :=BEGIN  !READ 1ST HALFWORD OF INSTRUCTION
        MAR← LOC;
        NBYTES ← 2;      !NO. OF BYTES TO BE READ
        INSTR ← 1 NEXT   !THIS MEMORY ACCESS IS DURING IFETCH
        MEMRD NEXT       !READ NBYTES FROM MEMORY
        IR<0:15> ← MBR<16:31> NEXT      !LOAD INSTRUCTION REGISTER
        CCOP←0           !INITIALIZE THE COND. CODE PARAMETER
END;  !IFETCH


IXQT := BEGIN            !DECODE OPCODE AND HANDLE INSTRUCTION
        (DECODE OP =>
\00     ILLINST;
\01     BALR;
\02     BTCR;
\03     BFCR;
\04     NR;
\05     CLR;
\06     ORinst;
\07     XR;
\08     LR;
\09     CR;
\0A     AR;
\0B     SR;
\0C     MHR;
\0D     DHR;
\0E     ILLINST;
\0F     ILLINST;


\10     SRLS;
\11     SLLS;
\12     CHVR;
\13     ILLINST;
\14     ILLINST;
\15     ILLINST;
\16     ILLINST;
\17     ILLINST;
\18     LPSWR;
\19     ILLINST;
\1A     ILLINST;
\1B     ILLINST;
\1C     MR;
\1D     DR;
\1E     ILLINST;
\1F     ILLINST;


\20     BTBS;
\21     BTFS;
\22     BFBS;
\23     BFFS;
\24     LIS;
\25     LCS;
\26     AIS;
\27     SIS;
\28     ler;
\29     cer;
\2A     aer;
\2B     ser;
\2C     mer;
\2D     der;
\2E     fxr;
\2F     flr;


\30     ILLINST;
\31     ILLINST;
\32     ILLINST;
\33     ILLINST;
\34     EXHR;
```

```
\35     ILLINST;
\36     ILLINST;
\37     ILLINST;
\38     ldr;
\39     cdr;
\3a     adr;
\3b     sdr;
\3c     mdr;
\3d     ddr;
\3e     fxdr;
\3f     fldr;

\40     STH;
\41     BAL;
\42     BTC;
\43     BFC;
\44     NH;
\45     CLH;
\46     OH;
\47     XH;
\48     LH;
\49     CH;
\4A     AH;
\4B     SH;
\4C     MH;
\4D     DH;
\4E     ILLINST;
\4F     ILLINST;

\50     ST;
\51     AM;
\52     ILLINST;
\53     ILLINST;
\54     Ninst;
\55     CL;
\56     O;
\57     X;
\58     Linst;
\59     Cinst;
\5A     Ainst;
\5B     S;
\5C     Minst;
\5D     Dinst;
\5E     crc12;
\5F     crc16;

\60     ste;
\61     ahm;
\62     ILLINST;
\63     lra;
\64     ATL;
\65     ABL;
\66     RTL;
\67     RBL;
\68     le;
\69     ce;
\6A     ae;
\6B     se;
\6C     me;
\6D     de;
\6E     ILLINST;
\6F     ILLINST;

\70     std;
\71     stme;
\72     lme;
\73     LHL;
\74     TBT;
\75     SBT;
\76     RBT;
```

```
\77     CBT;
\78     ld;
\79     cd;
\7a     ad;
\7b     sd;
\7c     md;
\7d     dd;
\7e     stmd;
\7f     lmd;


\80     ILLINST;
\81     ILLINST;
\82     ILLINST;
\83     ILLINST;
\84     ILLINST;
\85     ILLINST;
\86     ILLINST;
\87     ILLINST;
\88     ILLINST;
\89     ILLINST;
\8A     ILLINST;
\8B     ILLINST;
\8C     ILLINST;
\8D     ILLINST;
\8E     ILLINST;
\8F     ILLINST;

\90     SRHLS;
\91     SLHLS;
\92     STBR;
\93     LBR;
\94     EXBR;
\95     EPSR;
\96     wbr;
\97     rbr;
\98     whr;
\99     rhr;
\9A     wdr;
\9B     rdr;
\9C     ILLINST;
\9D     ssr;
\9E     ocr;
\9F     ILLINST;

\A0     ILLINST;
\A1     ILLINST;
\A2     ILLINST;
\A3     ILLINST;
\A4     ILLINST;
\A5     ILLINST;
\A6     ILLINST;
\A7     ILLINST;
\A8     ILLINST;
\A9     ILLINST;
\AA     ILLINST;
\AB     ILLINST;
\AC     ILLINST;
\AD     ILLINST;
\AE     ILLINST;
\AF     ILLINST;

\B0     ILLINST;
\B1     ILLINST;
\B2     ILLINST;
\B3     ILLINST;
\B4     ILLINST;
\B5     ILLINST;
\B6     ILLINST;
\B7     ILLINST;
\B8     ILLINST;
```

```
\B9     ILLINST;
\BA     ILLINST;
\BB     ILLINST;
\BC     ILLINST;
\BD     ILLINST;
\BE     ILLINST;
\BF     ILLINST;

\C0     BXH;
\C1     BXLE;
\C2     LPSW;
\C3     THI;
\C4     NHI;
\C5     CLHI;
\C6     OHI;
\C7     XHI;
\C8     LHI;
\C9     CHI;
\CA     AHI;
\CB     SHI;
\CC     SRHL;
\CD     SLHL;
\CE     SRHA;
\CF     SLHA;

\D0     STM;
\D1     LM;
\D2     STB;
\D3     LB;
\D4     CLB;
\D5     al;
\D6     wb;
\D7     rb;
\D8     wh;
\D9     rh;
\DA     wd;
\DB     rd;
\DC     ILLINST;
\DD     ss;
\DE     oc;
\DF     ILLINST;

\E0     TS;
\E1     SVC;
\E2     sint;
\E3     scp;
\E4     ILLINST;
\E5     ILLINST;
\E6     LA;
\E7     TLATE;
\E8     ILLINST;
\E9     ILLINST;
\EA     RRL;
\EB     RLL;
\EC     SRL;
\ED     SLL;
\EE     SRA;
\EF     SLA;

\F0     ILLINST;
\F1     ILLINST;
\F2     ILLINST;
\F3     TI;
\F4     NI;
\F5     CLI;
\F6     OI;
\F7     XI;
\F8     LI;
\F9     CI;
\FA     AI;
```

```
INTERDATA 8/32 ISP DESCRIPTION

\FB     SI;
\FC     ILLINST;
\FD     ILLINST;
\FE     ILLINST;
\FF     ILLINST
        )       !end opcode decode
END;  !IXQT


INTCHK:=BEGIN           !INTERRUPT CHECKING AND HANDLING
        (IF intvec =>                           !are there any pending i/o interrupts
        (DECODE ii =>                           !what is the processor's interrupt status
        \00 bailout emulate;                    !all interrupts are disabled
        \01 Begin
                (IFF set GEQ 3 THEN low←2;
                ELSE low←(set - 1)<2:0>
                ) Next
                (IF set EQL 0 => low←0)
           End;                                 !levels higher than the current reg. set are enabled
        \10 low←3;                              !all levels enabled
        \11 low←set                             !levels equal to or higher than the current reg. set are enabled
        ) Next  !end DECODE
                                                !low (4-bit temp) is the lowest interrupt level enabled
        intlev←0; temp4←1 Next                  !lets check for a level 0 interrupt
        getilev:=(IF (intlev LEQ low) AND (temp4 NEQ 0) =>                      !is intlev an enabled interrupt
                (IFF (temp4 AND intvec) THEN (intvec←intvec XOR temp4 Next ioint);
                                                !is there an interrupt pending of level intvec?
                                                !if so, clear the interrupt and process it
                        ELSE Begin
                        temp4←(temp4 ↑SL0 1)<3:0>;
                        intlev←(intlev + 1)<1:0> Next
                        getilev
                        End     !ELSE
                )       !end IFF
        )       !end getilev
        )       !end IF
End     !intchk


ERALCED
```

INTERDATA 8/32 ISP DESCRIPTION

```
!       MAIN PROGRAM FOLLOWS:

EMULATE :=BEGIN !EMULATION CYCLE
        (IF RUN =>
                (if NOT w =>    !is the wait bit on in the psww
                                !if so, just wait for ints
                IFETCH NEXT     !FETCH 1ST HALFWORD OF INSTRUCTION
                IXQT )   NEXT   !DECODE AND EXECUTE INSTRUCTION
                INTCHK NEXT     !CHECK AND HANDLE INTERRUPTS
                EMULATE !FETCH NEXT INSTRUCTION
        ) !end if
END !EMULATE
)       ! END INTERDATA
```

```
pdp11   :=
(DECLARE

!Please report errors to Dan Siewiorek, CMU, (412)-621-2600 x177
!
!The PDP-11 ISP has several features to aid in reading and data
!gathering. These include:
!
!1.) A word memory defined on top of a byte memory. Thus byte
!accesses can be separetly counted from word accesses.
!
!2.) All memory accessing is done through two routines, READ and WRITE
!on page 6-1. Memory management protection is enforced by these
!routines.
!
!3.) All effective address calculation is done by two routines called
!SOURCE (for the source operand) and DEST (for the destination
!operand). The routines are described on page 7-1.
!
!4.) All parameters are passed to procedures in the register called
!TEMP.
!
!5.) Instruction mnemonics are used as labels for the ISP sequence
!that simulates the instruction's effects. This provides easy reader
!reference as well as a counter for statistics gathering. Further,
!each mnemonic is followed by its value in the current decode
!statement (eg DIV\1).
!
!6.) Several types of labels were added both for statistics gathering
!and for control over the counters. For example, it was decided that
!condition code setting was a combinatorial action and that a register
!transfer should not be charged. Thus labels were added to condition
!code setting routines so that they could be opqued. The PDP-11
!architecture represented a good example why a simulator can do things
!in data collection that hand analysis would find difficult if not
!impossible to do. The R count of PDP-11 instructions was a function
!of the addressing mode used. Thus variables were added that counted
!the number of times source or destination addressing mode zero
!(register) was used with each instruction. Below is a key to the
!signivicance of the labels. <instruction> stands for the instruction
!mnemonic.
!
!
!↓_↓Label_↓\↓_Meaning_↓
!
!<instruction>\ISP to simulate instruction, used to count the number
!of times instruction was executed.
!
!I<instruction>\Multiple instructions defined by this procedure, such
!as MOV (move) and MOVB (move byte). Individual instruction labels
!internal to procedure.
!
!C<instruction>\Condition code setting portion of instruction
!
!cc<instruction>\Carry condition code
!
!cn<instruction>\Negative condition code
!
!cz<instruction>\Zero condition code
!
!cv<instruction>\oVerflow condition code
!
!d<instruction>\destination address mode = 0 (Register)
```

```
!
!s<instruction>\source address mode = 0 (Register)
!
!
!Following is a page by page description of the ISP:
!
!
!Page 2-1.  The primary memory and mappings (note word/byte memory and
!I/O  page),  central  processor registers,  and  the  floating  point
!processor status register.
!
!Page  3-1.   The  PDP-11/40 memory  management  registers  and  error
!registers that allow an instruction retry.
!
!Page  4-1.   Temporary  registers not  seen by  programmer.  These
!registers are necessary to completely define the algorithms performed
!by the hardware (such as address calculation) but these registers are
!not part of the architecture.
!
!Page 5-1.  Instruction decoding formats.
!
!Page 6-1.  Start of the procedures.  Memory accessing procedures.
!
!Page 7-1.  Effective address calculating procedures.
!
!Page 8-1.  Condition code setting procedures.
!
!Pages  9 through 15.  These are the  actual instruction definitions.
!Similar  instructions are  grouped together into  classes that follow
!the  several levels  of decoding that  the hardware  must go through.
!Since procedures must be defined before use, the decoding sequence is
!in reverse order from that encountered in the ISP.  As a guide to the
!reader,  the  following is a summary of the  decoding order.  Items in
!<>  represent instructions, all  others are just  procedure names for
!further decoding.
!
!.nofill;
!
!exec (main decode call execute)
!       reserop (reserved op code class)
!       <double operand instructions>
!       extop (extended instruction set)
!
!reserop
!       branop (branch instructions)
!       classop (secondary dscode into classes)
!
!extop
!       intext (integer extended instructions)
!       fpext (floating point instructions)
!
!branop
!       regop (register operations)
!       <branch instructions>
!
!regop
!       cpucon (cpu control instructions)
!       procon (program control instructions)
!
!classop
!       subemt (subroutine/emulator traps)
!       singlop (single operand instructions)
!       shiftop (shift instructions)
```

```
!
!fpext
!       fppcon (floating point processor mode control)
!       fsingl (floating point uniary instructions)
!       <floating point double operand instructions>
!
!fppcon
!       fsrcon (floating status register setting instructions)
!       <floating point processor mode control>
!
!Page 16-1.  The instruction interpretation cycle.
!
!
!It  should be pointed  out that  in the floating  point processor the
!precision  mode  bit  (single  or  double) determines  whether  the
!operation is performed with 32  or 64 bits.  Thus even though the ISP
!reads as  if there are two floating point  instruction sets, the same
!bit  pattern is interpreted as  both a single precision  and a double
!precision instruction as a function of the precision mode bit.
```

```
MACRO BEGIN     := ( $
MACRO END       := ) $

!mp state
!---------

mb[#167777:0]<7:0>;      !the addressing space (28K)
mbio[#777777:#770000]<7:0>;      !the I/o page (4K)
        mw[#73777:0]<15:0>              := mb[#167777:0]<7:0>;
!       mb[#167777:0]<7:0>              := m[#167777:0]<7:0>;
        mwio[#377777:#374000]<15:0>     := mbio[#777777:#770000]<7:0>;
!       mbio[#777777:#770000]<7:0>      := mio[#777777:#770000]<7:0>;

mar\memory.addr.reg<17:0>;
        wmar\word.mar<17:1>             := mar<17:1>;

mbr\memory.buff.reg<15:0>;
        bmbr\byte.mbr<7:0>              := mbr<7:0>;

shsign\ash.instruction.offset.sign<>    := mbr<5>;

shval\ash.instruction.offset.value<4:0> := mbr<4:0>;



!pc state
!---------


r\register[7:0]<15:0>           := mwio[#374007:#374000]<15:0>;
        sp<15:0>                := r[6]<15:0>;  ! stack pointer
        pc<15:0>                := r[7]<15:0>;  ! program counter

ps<15:0>;               ! program status word
        cm\current.mode<1:0>    := ps<15:14>;   ! current address space (kernel/user)
        p\priority<2:0>         := ps<7:5>;     ! current process priority
        t\trace<>               := ps<4>;
        cc\condition.codes<3:0> := ps<3:0>;
                n\negative<>            := cc<3>;
                z\zero<>                := cc<2>;
                v\overflow<>            := cc<1>;
                c\carry<>               := cc<0>;

a\activity<0:1>;
        macro run       := (a eql 0)$
        macro wait      := (a eql 1)$
        macro halt      := (a eql 2)$

!floating point processor state
!------------------------------

fppsr\fpp.status.register<15:0>;
        fer<>   := fppsr<15>;           !floating point error flag
        fid<>   := fppsr<14>;           !floating interrupt disable
        fiuv<>  := fppsr<11>;           !floating interrupt on
                                        !undefined variables enable flag

        fiu<>   := fppsr<10>;           !floating interrupt on underflow
                                        !enable flag

        fiv<>   := fppsr<9>;            !floating interrupt on overflow
                                        !enable flag
```

```
        fic<>   := fppsr<8>;        !floating interrupt on integer
                                    !conversion error enable flag

        fd<>    := fppsr<7>;        !floating precision, one implies
                                    !double precision, zero single

        fi<>    := fppsr<6>;        !interger precision for interger
                                    !to floating conversions.  One
                                    !implies double precision, zero
                                    !single.

        ft<>    := fppsr<5>;        !truncation or round result.
                                    !One implies truncation, zero
                                    !rounding.

        fmm<>   := fppsr<4>;        !maintenance mode
        fn<>    := fppsr<3>;        !floating negative condition code
        fz<>    := fppsr<2>;        !floating zero condition code
        fv<>    := fppsr<1>;        !floating overflow condition code
        fc<>    := fppsr<0>;        !floating carry condition code
```

```
!memory management
!------------------

        par\page.address.register[15:0]<15:0>          := mmio[#377737:#377720]<15:0>;
        pdr\page.description.register[15:0]<15:0>      := mmio[#377717:#377700]<15:0>;

        macro paf\page.address.field    := 11:0$
        macro acf\access.control.field  := 2:1$
        macro ed\expansion.direction    := 3$
        macro wbit\written.bit          := 6$
        macro plf\page.length.field     := 14:8$

        sr0\status.register.0<15:0>     := mmio[#377675]<15:0>;
                anr\abort.nonresident.flag<>    := sr0<15>;
                aple\abort.page.length.flag<>   := sr0<14>;
                aro\abort.read.only.flag<>      := sr0<13>;
                am\abort.mode<1:0>              := sr0<6:5>;
                apn\abort.page.number<2:0>      := sr0<3:1>;
                emm\enable.memory.management<>  := sr0<0>;

        sr2\status.register.2<15:0>     := mmio[#377677]<15:0>;
```

```
!error flags and temporary registers
!-------------------------------------

boundary.error<>;
stack.overflow<>;
illegal.instruction<>;
byop\byte.read.flag<>;
sbyop\save.area.for.byop<>;
src<17:0>;
dst<17:0>;
temp<17:0>;
temp1<3:0>;
macro dcond := IF dmodd EQL #0$
macro scond := IF smosd EQL #0$
temp2<32:0>;
```

PDP-11 ISP DESCRIPTION 5-1

```
!instruction format
!------------------

i\instruction<15:0>;
        bop\binary.operation<2:0>       := i<14:12>;

ir\instruction.register<15:0>:=i<15:0>;

! source addressing information
        s\source.field<5:0>             := i<11:6>;
                sm\source.mode<1:0>             := s<5:4>;
                sd\source.deferred<>            := s<3>;
                sr\source.register<2:0>         := s<2:0>;

        macro sr67      := (sr<2:1> eql #3)$

! destination addressing information
        d\destination.field<5:0>        := i<5:0>;
                dm\destination.mode<1:0>        := d<5:4>;
                dd\destination.deferred<>       := d<3>;
                dr\destination.register<2:0>    := d<2:0>;

        macro dr67      := (dr<2:1> eql #3)$

        uop\unary.operation<2:0>        := i<8:6>;
        offset<7:0>                     := ir<7:0>;
        rop\register.operation<1:0>     := i<7:6>;
        jetop\jsr.emulator.trap.op<>    :=i<15>;
        etop\emulator.trap.op<>         := i<8>;
        concop\condition.code.op<10:0>  := i<15:5>;
        cpuop\cpu.control.op<2:0>       := i<2:0>;
        contop\cpu.control.class.op<2:0>:= i<5:3>;
        brop\branch.op.code<2:0>        := i<10:8>;
        intop\extended.integer.op<2:0>  := i<11:9>;
        typeop\class.op.code.bits<1:0>  := i<10:9>;
        resop\reserve.op<>              := i<11>;
        ccop\condition.code.second.op<> := i<4>;

! floating point instruction decoding

        fbop\floating.binary.operation<3:0>:= i<11:8>;
        fuop\floating.unary.operation<1:0> := i<7:6>;
        fmsop\floating.mode.setting.op<1:0>:= i<1:0>;
        fdsop\floating.double.single.mode.setting.op<>
                                        := i<4>;


! end of register declarations
```

```
!functional declarations
!-----------------------

abort :=
        BEGIN
        am ← cm; apn ← mar<15:13> NEXT
        pc ← #250
        END;

read :=
        BEGIN
        (IF emm =>
                (DECODE cm => temp1 ← mar<15:13>; abort; abort; temp1 ← (mar<15:13>+8)<3:0>) NEXT
                mar ← ((par[temp1]<paf> + mar<12:6>)<11:0>) @ mar<5:0> NEXT
                (IF not pdr[temp1]<acf> => abort; anr ← 1) NEXT
                (IF (mar<12:6> gtr pdr[temp1]<plf>) and NOT pdr[temp1]<ed> => abort; apie ← 1) NEXT
                (IF (mar<12:6> lss pdr[temp1]<plf>) and  pdr[temp1]<ed> => abort; apie ← 1)
                ) NEXT
        (IF mar<15:13> EQL #7 => mar<17:16>←#3) NEXT    ! map into io page
        (DECODE mar<17:13> eql #37 =>
        \no     (DECODE byop => mbr ← mw[wmar]; mbr ← mb[mar]);
        \yes    (DECODE byop => mbr ← mwio[wmar]; mbr ← mbio[mar])
                )
        END;

write :=
        BEGIN
        (IF emm =>
                (DECODE cm => temp1 ← mar<15:13>; abort; abort; temp1 ← (mar<15:13>+8)<3:0>) NEXT
                mar ← ((par[temp1]<paf> + mar<12:6>)<11:0>) @ mar<5:0> NEXT
                (IF pdr[temp1]<acf> eql 0 => abort; anr ← 1) NEXT
                (IF pdr[temp1]<acf> eql 1 => abort; aro ← 1) NEXT
                (IF (mar<12:6> gtr pdr[temp1]<plf>) and NOT pdr[temp1]<ed> => abort; apie ← 1) NEXT
                (IF (mar<12:6> lss pdr[temp1]<plf>) and  pdr[temp1]<ed> => abort; apie ← 1) NEXT
                pdr[temp1]<wbit> ← 1
                ) NEXT
        (IF MAR<15:13> EQL #7 => mar<17:16>←#3) NEXT
        (DECODE mar<17:13> eql #37 =>
        \no     (DECODE byop => mw[wmar] ← mbr; mb[mar] ← bmbr);
        \yes    (DECODE byop => mwio[wmar] ← mbr; mbio[mar] ← bmbr)
                )
        END;

bus.reset := (pc ← pc);

nop := (temp ← temp);
```

```
!operand determination
!--------------------
!source loads the value of the source operand into register src.
!dest loads the address of the destination operand into register dst
!and fetches the operand to the mbr.


source :=
        BEGIN
        (DECODE sm =>
        ASREG\0:=(src ← r[sr] NEXT
                (DECODE sd =>
                \0      SREG:=nop;
                \1      SREGD:=nop)
                );

        ASINC\1:=BEGIN
                (DECODE sd =>
                \0      (DECODE byop =>
                        \0      SINC:=nop;
                        \1      SINCB:=nop
                        );
                \1      SINCD:=nop
                ) NEXT
                mar ← r[sr] NEXT
                (DECODE sr67 or sd => r[sr] ← (r[sr]+(2-byop))<15:0>; r[sr] ← (r[sr]+2)<15:0>) NEXT
                 read NEXT
                src ← mbr
                END;

        ASDEC\2:=BEGIN
                (DECODE sd =>
                \0      (DECODE byop =>
                        \0      SDEC:=nop;
                        \1      SDECB:=nop
                        );
                \1      SDECD:=nop
                ) NEXT
                (DECODE sr67 or sd => r[sr] ← (r[sr] - (2-byop))<15:0>; r[sr] ← (r[sr] -2)<15:0>) NEXT
                mar ← r[sr] NEXT
                read NEXT
                src ← mbr
                END;

        ASIND\3:=BEGIN
                (DECODE sd =>
                \0      SIND:=nop;
                \1      SINDD:=nop
                ) NEXT
                mar ← pc NEXT
                pc ← (pc + 2)<15:0> NEXT
                read NEXT
                mar ← (mbr + r[sr])<15:0> NEXT
                read NEXT
                src ← mbr
                END
                ) NEXT

        (IF sd => mar ← src  NEXT read NEXT src ← mbr )
        END;
```



```
dest :=
        BEGIN
```

```
        (DECODE dm =>
                                !general registers have addresses 777700:777717
        ADREG\0:=(dst ← (#37400 @ dr)@0 NEXT
                (DECODE dd =>
                \0      DREG:=nop;
                \1      DREGD:=nop)
                );

        ADINCR\1:=BEGIN
                (DECODE dd =>
                \0      (DECODE byop =>
                        \0      DINCR:=nop;
                        \1      DINCRB:=nop
                        );
                \1      DINCRD:=nop
                ) NEXT
                dst ← r[dr] NEXT
                (DECODE dr67 or dd => r[dr] ← (r[dr]+(2-byop))<15:0>; r[dr] ← (r[dr]+2)<15:0>)
                END;

        ADDECR\2:=BEGIN
                (DECODE dd =>
                \0      (DECODE byop =>
                        \0      DDECR:=nop;
                        \1      DDECRB:=nop
                        );
                \1      DDECRD:=nop
                ) NEXT
                (DECODE dr67 or dd => r[dr] ← (r[dr]-(2-byop))<15:0>; r[dr] ← (r[dr]-2)<15:0>) NEXT
                dst ← r[dr]
                END;

        ADIND\3:=BEGIN
                (DECODE dd =>
                \0      DIND:=nop;
                \1      DINDD:=nop
                ) NEXT
                mar ← pc NEXT
                pc ← (pc + 2)<15:0> NEXT
                read NEXT
                dst ← (mbr + r[dr])<15:0>
                END
                ) NEXT

        mar ← dst NEXT

        (IF dd => sbyop ← byop NEXT byop ← 0 NEXT read NEXT byop ← sbyop; dst ← mbr; mar ← mbr)
        END;
```

```
!condition code setting and branch operations
!-------------------------------------------

setncc\set.n.condition.code:=
        (DECODE byop => n ← temp<15>; n ← temp<7>);

setvcc\set.v.condition.code:=
        (DECODE byop => v ← (temp<15:0> eql #100000); v ← (temp<7:0> eql #200) );

setzcc\set.z.condition.code:=
        (DECODE byop => z ← (temp<15:0> eql 0); z ← (temp<7:0> eql 0) );

signextend:=
        (DECODE offset<7> => temp ← offset; temp ← #377 @ offset);

branch:=
        BEGIN
        signextend NEXT
        pc ← (pc + (temp ^* 2 ))<15:0>
        END;
```

```
subemt\subroutine.emulator.trap.and.trap.instructions:=
        BEGIN
        DECODE jetop =>
! jump to subroutine, jsr op code #004
        JSR\0:= BEGIN
                dest NEXT
                temp ← mar NEXT
                sp ← (sp - 2)<15:0> NEXT
                mar ← sp; mbr ← r[sr] NEXT
                write NEXT
                r[sr] ← pc NEXT
                pc ← temp<15:0>
                END;

        \1      BEGIN
                DECODE i<8> =>
        ! emulator trap op codes, op code #104000:#104377
                EMUL\0:=BEGIN
                        byop ← 0; sp ← (sp - 2)<15:0>; mbr ← ps NEXT
                        write NEXT
                        sp ← (sp - 2)<15:0>; mbr ← pc NEXT
                        write NEXT
                        mar ← #30 NEXT
                        read NEXT
                        pc ← mbr NEXT
                        mar ← #32 NEXT
                        read NEXT
                        ps ← mbr
                        END;

        ! trap op codes, op code #104400:#104777
                TRAP\1:=BEGIN
                        byop ← 0; sp ← (sp - 2)<15:0>; mbr ← ps NEXT
                        write NEXT
                        sp ← (sp - 2)<15:0>; mbr ← pc NEXT
                        write NEXT
                        mar ← #34 NEXT
                        read NEXT
                        pc ← mbr NEXT
                        mar ← #36 NEXT
                        read NEXT
                        ps ← mbr
                        END
                END
        END;


singlop\single.operand.instructions:=
        BEGIN
        DECODE uop =>

! clear and clear byte, clr op code #0050, clrb op code #1050
        ICLR\0:=        BEGIN
                (DECODE byop =>
                        ( CLR:=nop NEXT (dcond => (dclr:=nop)));
                        ( CLRB:=nop NEXT (dcond => (dclrb:=nop)))
                ) NEXT
ccir:=          (cc ← '0100) NEXT
                dest NEXT
                mbr ← 0 NEXT
                write
                END;
```

```
! complement and complement byte, com op code #0051, comb op code #1051
        ICOM\1:=        BEGIN
                (DECODE byop =>
                        ( COM:=nop NEXT (dcond => (dcom:=nop)));
                        ( COMB:=nop NEXT (dcond => (dcomb:=nop)))
                ) NEXT
                dest NEXT
                read NEXT
                temp ← not mbr NEXT
ccom:=          (v←0 NEXT
                c←1) NEXT
                setncc NEXT
                setzcc NEXT
                mbr ← temp<15:0> NEXT
                write
                END;

! increment and increment byte, inc op code #0052, incb op code #1052
        IINC\2:=        BEGIN
                (DECODE byop =>
                        ( INC:=nop NEXT (dcond => (dinc:=nop)));
                        ( INCB:=nop NEXT (dcond => (dincb:=nop)))
                ) NEXT
                dest NEXT
                read NEXT
                temp ← mbr + 1 NEXT
                setvcc NEXT
                setncc NEXT
                setzcc NEXT
                mbr ← temp<15:0> NEXT
                write
                END;

! decrement and decrement byte, dec op code #0053, decb op code #1053
        IDEC\3:=        BEGIN
                (DECODE byop =>
                        ( DEC:=nop NEXT (dcond => (ddec:=nop)));
                        ( DECB:=nop NEXT (dcond => (ddecb:=nop)))
                ) NEXT
                dest NEXT
                read NEXT
                temp ← mbr - 1 NEXT
                setvcc NEXT
                setncc NEXT
                setzcc NEXT
                mbr ← temp<15:0> NEXT
                write
                END;

! negate and negate byte, neg op code #0054, negb op code #1054
        INEG\4:=        BEGIN
                (DECODE byop =>
                        ( NEG:=nop NEXT (dcond => (dneg:=nop)));
                        ( NEGB:=nop NEXT (dcond => (dnegb:=nop)))
                ) NEXT
                dest NEXT
                read NEXT
                temp ← (not mbr) + 1 NEXT
                setvcc NEXT
                setncc NEXT
                setzcc NEXT
cneg:=          (c ← (temp<15:0> neq 0)) NEXT
```

```
                mbr ← temp<15:0> NEXT
                write
                END;

! add carry and add carry byte, adc op code #0055, adcb op code #1055
        IADC\5:=        BEGIN
                (DECODE byop =>
                        ( ADC:=nop NEXT (dcond => (dadc:=nop)));
                        ( ADCB:=nop NEXT (dcond => (dadcb:=nop)))
                ) NEXT
                dest NEXT
                read NEXT
                temp ← mbr + c NEXT
cadc:=          (DECODE byop =>
                \0      BEGIN
                        v ← (temp<15:0> eql #100000) and c NEXT
                        c ← ( (temp<15:0> eql 0) and c )
                        END;
                \1      BEGIN
                        v ← (temp<7:0> eql #200) and c NEXT
                        c ← ((temp<7:0> eql 0) and c )
                        END
                        ) NEXT
                setncc NEXT
                setzcc NEXT
                mbr ← temp<15:0> NEXT
                write
                END;

! subtract and subtract carry byte, sbc op code #0056, sbcb op code #1056
        ISBC\6:=        BEGIN
                (DECODE byop =>
                        ( SBC:=nop NEXT (dcond => (dsbc:=nop)));
                        ( SBCB:=nop NEXT (dcond => (dsbcb:=nop)))
                ) NEXT
                dest NEXT
                read NEXT
                temp ← mbr - c NEXT
csbc:=          (DECODE byop =>
                        c ← ( (temp<15:0> eql #177777) and c );
                        c ← ( (temp<7:0> eql #377) and c )
                        ) NEXT
                setvcc NEXT
                setncc NEXT
                setzcc NEXT
                mbr ← temp<15:0> NEXT
                write
                END;

! test and test byte, tst op code #0057, tstb op code #1057
        ITEST\7:=BEGIN
                (DECODE byop =>
                        ( TEST:=nop NEXT (dcond => (dtst:=nop)));
                        ( TESTB:=nop NEXT (dcond => (dtstb:=nop)))
                ) NEXT
                dest NEXT
                read NEXT
                temp ← mbr NEXT
ctst:=          (v←0 NEXT
                c←0) NEXT
                setncc NEXT
                setzcc
                END
```

AD-A049 483 ARMY ELECTRONICS COMMAND FORT MONMOUTH N J F/G 9/2
COMPUTER FAMILY ARCHITECTURE SELECTION COMMITTEE FINAL REPORT. --ETC(U)
SEP 77 M BARBACCI, R GORDON, R HOWBRIGG

UNCLASSIFIED ECOM-4529 NL

3 OF 3
AD
A049483

END
DATE
FILMED
3-78
DDC

```
        END;

shiftop\shift.instructions:=
        BEGIN
        DECODE uop =>
! rotate right and rotate right byte, ror op code #0060, rorb op code #1060
        IROR\0:=        BEGIN
                (DECODE byop =>
                        ( ROR:=nop NEXT (dcond => (dror:=nop)));
                        ( RORB:=nop NEXT (dcond => (drorb:=nop)))
                ) NEXT
                dest NEXT
                read NEXT
                temp ← mbr NEXT
                (DECODE byop =>
                \0      (temp<16:0> ← (c @ temp<15:0>) frr 1 NEXT cror:=( c ← temp<16>); mbr ← temp<15:0>);
                \1      (temp<8:0> ← (c @ temp<7:0>) frr 1 NEXT crorb:= ( c ← temp<8>); bmbr ← temp<7:0>)
                        ) NEXT
                setncc NEXT
                setzcc NEXT
cvror:=         (v ← n xor c) NEXT
                write
                END;

! rotate left and rotate left byte, rol op code #0061, rolb op code #1061
        IROL\1:=        BEGIN
                (DECODE byop =>
                        ( ROL:=nop NEXT (dcond => (drol:=nop)));
                        ( ROLB:=nop NEXT (dcond => (drolb:=nop)))
                ) NEXT
                dest NEXT
                read NEXT
                temp ← mbr NEXT
                (DECODE byop =>
                        (temp<16:0> ← (c @ temp<15:0>) frl 1 NEXT crol:= ( c ← temp<16>); mbr ← temp<15:0>);
                        (temp<8:0> ← (c @ temp<7:0>) frl 1 NEXT crolb:= ( c ← temp<8>); bmbr ← temp<7:0>)
                        ) NEXT
                setncc NEXT
                setzcc NEXT
cvrol:=         (v ← n xor c) NEXT
                write
                END;

! arithmetic shift right and arithmetic shift right byte, asr op code #0062, asrb op code #1062
        IASR\2:=        BEGIN
                (DECODE byop =>
                        ( ASR:=nop NEXT (dcond => (dasr:=nop)));
                        ( ASRB:=nop NEXT (dcond => (dasrb:=nop)))
                ) NEXT
                dest NEXT
                read NEXT
                temp ← mbr NEXT
casr:=          (c ← temp<0>) NEXT
                (DECODE byop =>
                        (temp<15:0> ← (temp<15:0>) fsr temp<15> NEXT  mbr ← temp<15:0>);
                        (temp<7:0> ← (temp<7:0>) fsr temp<7> NEXT mbr ← temp<8:1>)
                        ) NEXT
                setncc NEXT
                setzcc NEXT
cvasr:=         (v ← n xor c) NEXT
                write
                END;
```

```
! arithmetic shift left and arithmetic shift left byte, asl op code #0063, aslb op code #1063
        IASL\3:=        BEGIN
                (DECODE byop =>
                        ( ASL:=nop NEXT (dcond => (dasl:=nop)));
                        ( ASLB:=nop NEXT (dcond => (daslb:=nop)))
                ) NEXT
                dest NEXT
                read NEXT
                temp ← mbr NEXT
                (DECODE byop =>
                        (temp<16:0> ← (c @ temp<15:0>) fsl 0 NEXT casl:=( c ← temp<16>); mbr ← temp<15:0>);
                        (temp<8:0> ← (c @ temp<7:0>) fsl 0 NEXT caslb:=( c ← temp<8>); mbr ← temp<7:0>)
                        ) NEXT
                setncc NEXT
                setzcc NEXT
cvasl:=         (v ← n xor c) NEXT
                write
                END;

! mark and unused op codes, mark op code #0064
        MARK\4:=BEGIN
                IF not jetop =>
                        sp ← (sp + (d fsl 0))<15:0> NEXT pc ← r[5] NEXT
                        mar ← sp NEXT
                        read NEXT
                        r[5] ← mbr; sp ← (sp + 2)<15:0>
                END;

! move from previous instruction and data space, mfpi op code #0065, mfpd op code #1065
! 11/70 instructions
        MFP\5:= ((dcond => (smfp:=nop)) NEXT
                (DECODE jetop => nop;nop));

! move to previous instruction and data space, mtpi op code #0066, mtpd op code #1066
! 11/70 instructions
        MTP\6:= ((dcond => (smtp:=nop)) NEXT
                (DECODE jetop => nop; nop));

! sign extend and unused op code, sxt op code #0067
        SXT\7:= BEGIN
                IF not jetop =>
                        (dcond => (dsxt:=nop)) NEXT
                        dest NEXT
                        read NEXT
                        (DECODE n => mbr ← 0; mbr ← #177777) NEXT
csxt:=                  (z ← not n; v ← 0) NEXT
                        write
                END
        END;
```

```
        ! condition code operators. selectively clears
        ! or sets the specified condition code.
        ! the assembler recognizes the mnemonics
        ! clc, clv, clz, cln, ccc (for clear all
        ! condition codes), sec, sev, sez, sen, and scc.
        ! compound setting or clearing is accomplished
        !  by oring.
cco :=
        BEGIN
        IF (concop eql #0005) =>
ccco:=          (DECODE ccop =>
                \0      cc ← cc and not i<3:0>;
                \1      cc ← cc or i<3:0>
                )
        END;


cpucon\cpu.control.instructions :=
        BEGIN
        IF contop eql 0 =>
                BEGIN
                DECODE cpuop =>
                ! halt, halt op code #000000
                HLT\0:= a ← 2;

                ! wait for interrupt, wait op code #000001
                WAIT\1:=a ← 1;

                ! return from interrupt, rti op code #000002
                RTI\2:= BEGIN
                        mar ← sp NEXT
                        read NEXT
                        pc ← mbr; sp ← (sp + 2)<15:0> NEXT
                        mar ← sp NEXT
                        read NEXT
                        ps ← mbr; sp ← (sp + 2)<15:0>
                        END;

                ! breakpoint trap, bpt op code #000003
                BPT\3:= BEGIN
                        sp ← (sp - 2)<15:0>; mbr ← ps NEXT
                        write NEXT
                        sp ← (sp - 2)<15:0>; mbr ← pc NEXT
                        write NEXT
                        mar ← #14 NEXT
                        read NEXT
                        pc ← mbr NEXT
                        mar ← #16 NEXT
                        read NEXT
                        ps ← mbr
                        END;

                ! input/output trap, iot op code #000004
                IOT\4:= BEGIN
                        sp ← (sp - 2)<15:0>; mbr ← ps NEXT
                        write NEXT
                        sp ← (sp - 2)<15:0>; mbr ← pc NEXT
                        write NEXT
                        mar ← #20 NEXT
                        read NEXT
                        pc ← mbr NEXT
                        mar ← #22 NEXT
```

```
                        read NEXT
                        ps ← mbr
                        END;

                ! reset external bus, reset op code #000005
                RSET\5:=bus.reset;

                ! return from trap, rtt op code #000006
                RTT\6:= BEGIN
                        mar ← sp NEXT
                        read NEXT
                        pc ← mbr; sp ← (sp + 2)<15:0> NEXT
                        mar ← sp NEXT
                        read NEXT
                        ps ← mbr; sp ← (sp + 2)<15:0>
                        END;

                ! unused op code
                \7      nop
                END
        END;

procon\program.control.instructions :=
        BEGIN
        DECODE contop =>
        ! return from subroutine, rts op code #00020
        RTS\0:= BEGIN
                pc ← r[dr] NEXT
                mar ← sp NEXT
                read NEXT
                sp ← (sp + 2)<15:0> NEXT
                r[dr] ← mbr
                END;

        ! unused op code
        \1      nop;

        ! unused op code
        \2      nop;

        ! set priority level, spl op code #00023
        ! 11/70 instruction
        SPL\3:= nop;

        ! NEXT four op codes are condition code setting
        \4      cco;

        \5      cco;

        \6      cco;

        \7      cco
        END;
```

```
regop\register.operations:=
        BEGIN
        DECODE rop =>
        ! cpu control instructions
        \0      cpucon;

        ! jump, jmp op code #0001
        JMP\1:= BEGIN
                dest NEXT
                pc ← mar<15:0>
                END;

        ! program control instructions
        \2      procon;

        ! swap bytes, swab op code #0003
        SWAB\3:=BEGIN
                (dcond => (dswab:=nop)) NEXT
                dest NEXT
                read NEXT
                temp ← bmbr @ mbr<15:8> NEXT
cswap:=         (n ← temp<7>; z ← (temp<7:0> eql 0);
                v ← 0; c ← 0) NEXT
                write
                END

        END;
```

```
branop\branch.op.codes:=
        BEGIN
        DECODE jstop @ brop =>
! register instructions
        \0      regop;
! branch, br op code #0004
        BR\1:=  branch;
! branch IF not equal, bne op code #0010
        BNE\2:= (IF not z => branch);

! branch IF equal, beq op code #0014
        BEQ\3:= (IF z => branch);

! branch IF greater than or equal, bge op code #0020
        BGE\4:= (IF not (n xor v) => branch);

! branch IF less than, blt op code #0024
        BLT\5:= (IF (n xor v) => branch);

! branch IF greater than, bgt op code #0030
        BGT\6:= (IF not (z or (n xor v)) => branch);

! branch IF less than or equal, ble op code #0034
        BLE\7:= (IF (z or (n xor v)) => branch);

! branch IF plus, bpl op code #1000
        BPL\10:=(IF not n => branch);

! branch IF minus, bmi op code #1004
        BMI\11:=(IF n => branch);

! branch IF higher, bhi op code #1010
        BHI\12:=(IF (not c) and (not z) => branch);

! branch IF lower or same, blos op code #1014
        BLOS\13:=(IF (c or z) => branch);

! branch IF overflow clear, bvc op code #1020
        BVC\14:=(IF not v => branch);

! branch IF overflow set, bvs op code #1024
        BVS\15:=(IF v => branch);

! branch IF carry clear, bcc op code #1030
        BCC\16:=(IF not c => branch);

! branch IF carry set, bcs op code #1034
        BCS\17:=(IF c => branch)
        END;
```

```
intext\integer.extended.op.codes:=
        BEGIN
        DECODE intop =>
        ! integer multiply, mul op code #070
        ! 11/40 extended instruction
        MUL\0:= BEGIN
                (dcond => (smul:=nop)) NEXT
                dest NEXT
                read NEXT
                (DECODE mbr<15> =>
                \0      (DECODE r[sr]<15> =>
                        \0      temp2←(mbr*r[sr])<31:0>;
                        \1      (temp2←(mbr*(MINUS r[sr]))<31:0> NEXT
                                temp2←(MINUS temp2)<32:0>)
                        );
                \1      (DECODE r[sr]<15> =>
                        \0      (temp2←((MINUS mbr)*r[sr])<31:0> NEXT
                                temp2←(MINUS temp2)<32:0>);
                        \1      temp2←((MINUS mbr)*(MINUS r[sr]))<31:0>
                        )
                ) NEXT
cnmul:=         (n←temp2<31>) NEXT
czmul:=         (z←(temp2<31:0> eql 0)) NEXT
cvmul:=         (v←0) NEXT
ccmul:=         (c←(temp2<31:16> NEQ #177777) AND (temp2<31:16> NEQ 0)) NEXT
                (DECODE sr<0> =>
                \0      (r[sr]←temp2<31:16> NEXT
                        r[sr OR #1]←temp2<15:0>);
                \1      r[sr]←temp2<15:0>
                )
                END;

        ! integer divide, div op code #071
        ! 11/40 extended instruction
        DIV\1:= BEGIN
                (dcond=>(sdiv:=nop)) NEXT
                dest NEXT
                read NEXT
                ( IF (mbr NEQ 0) =>
                  (DECODE mbr<15> =>
                  \0    (DECODE r[sr]<15> =>
                        \0      (temp2←(r[sr]@r[sr OR #1])<31:0>/mbr NEXT
                                r[sr OR #1]←(r[sr]@r[sr OR #1]-temp2<15:0>*mbr)<15:0> NEXT
                                v←(r[sr] GTR mbr) );
                        \1      (temp2←(MINUS r[sr]@r[sr OR #1])<31:0>/mbr NEXT
                                r[sr OR #1] ← (MINUS(MINUS r[sr]@r[sr OR #1] - temp2<15:0>*mbr))<15:0> NEXT
                                temp2←(MINUS temp2)<32:0> NEXT
                                v←((MINUS r[sr]) GTR mbr) )
                        );
                  \1    (DECODE r[sr]<15> =>
                        \0      (temp2←(r[sr]@r[sr OR #1])<31:0>/(MINUS mbr) NEXT
                                r[sr OR #1] ← (MINUS (r[sr]@r[sr OR #1]  -  temp2<15:0>*(MINUS mbr)))<15:0>
                         NEXT   temp2←(MINUS temp2)<37:0> NEXT
                                v←(r[sr] GTR (MINUS mbr)) );
                        \1      (temp2← (MINUS r[sr]@r[sr OR #1])<31:0>/(MINUS mbr) NEXT
                         NEXT   r[sr OR #1] ← (MINUS r[sr]@r[sr OR #1]  -  'emp2<15:0>*(MINUS mbr) )<15:0>
                                v←( MINUS r[sr]  GTR  MINUS mbr ) )
                        )
                ) NEXT
cndiv:=         (n←temp2<31>) NEXT
czdiv:=         (z←(temp2<31:0> EQL 0))
                ) NEXT
```

```
cvdiv:=         (v←(mbr EQL 0) OR v) NEXT
ccdiv:=         (c←(mbr EQL 0)) NEXT
                r[sr]←temp2<15:0>
                END;

        ! shift arithmetically, ash op code #072
        ASH\2:= BEGIN
                (dcond => (dash:=nop)) NEXT
                dest NEXT
                read NEXT
                temp ← r[sr] NEXT
                BEGIN
                        DECODE shsign =>
                        \0      (temp<16:0>←(c@temp<15:0>) fsl0 shval NEXT
cash:=                                  ( c←temp<16>));
                        \1      ((DECODE temp<15> =>
                                \0      temp<15:0>←temp<15:0> fsr0 ((not shval)+1);
                                \1      temp<15:0>←temp<15:0> fsr1 ((not shval)+1)) NEXT
cashb:=                         (c←temp<8>))
                END NEXT
                setncc NEXT
                setzcc NEXT
cvash:=         (v←temp<15> xor mbr<15>) NEXT
                r[sr]←temp<15:0>
                END;

        ! arithmetical shift combined, ashc op code #073
        ! 11/70 extended instruction
        ASHC\3:=        BEGIN
                (dcond => (dashc:=nop)) NEXT
                dest NEXT
                read NEXT
                temp ← r[sr] NEXT
                (DECODE sr<0> =>
        \0      BEGIN
                        DECODE shsign =>
                        \0      (temp2<32:0>←(c@(temp<15:0>@r[sr OR #1])) fsl0 shval NEXT
cashc2:=                                (c←temp2<32>));
                        \1      ((DECODE temp<15> =>
                                \0      temp2<31:0>←(temp<15:0>@r[sr OR #1]) fsr0 ((not shval)+1);
                                \1      temp2<31:0>←(temp<15:0>@r[sr OR #1]) fsr1 ((not shval)+1)) NEXT
cashcb:=                        (c←temp2<8>))
                END;
        \1      BEGIN
                        DECODE shsign =>
                        \0      (temp2<16:0>←(c@temp<15:0>) fsl0 shval NEXT
cashc1:=                                ( c←temp2<16>));
                        \1      ((DECODE temp<15> =>
                                \0      temp2<15:0>←temp<15:0> fsr0 ((not shval)+1);
                                \1      temp2<15:0>←temp<15:0> fsr1 ((not shval)+1)) NEXT
cashhc:=                                (c←temp2<8>))
                END) NEXT
cnashc:=        (DECODE sr<0> => n←temp2<31>; n←temp2<15>) NEXT
czashc:=        (DECODE sr<0> => z←(temp2<31:0> eql 0); z←(temp2<15:0> eql 0) ) NEXT
cvashc:=        (DECODE sr<0> => v←(temp2<31> xor mbr<15>); v←(temp2<15> xor mbr<15>)) NEXT
                (DECODE sr<0> => (r[sr]←temp2<31:16> NEXT r[sr OR #1]←temp2<15:0>); (r[sr]←temp2<15:0>))
                END;

        ! exclusive or, xor op code #074
        EXOR\4:=BEGIN
                (dcond => (dexor:=nop)) NEXT
                dest NEXT
                read NEXT
```

```
                temp ← r[sr] xor mbr NEXT
                mbr ← temp NEXT
                setncc NEXT
                setzcc NEXT
cexor:=         (v ← 0) NEXT
                write
                END;

        ! remaining instructions 11/40 floating point or unused op codes

        \5      nop;

        \6      nop;

        SOB\7:= BEGIN
                r[sr] ← (r[sr] - 1)<15:0> NEXT
                (IF r[sr] neq 0 => pc ←  (pc - ir<5:0> tsl 0)<15:0>)
                END
        END;

fsrcon\floating.point.processor.mode.control :=
        (DECODE fdsop =>
        \0      BEGIN
                DECODE fmsop =>

                ! copy floating condition codes, cfcc
                ! op code #170000
                cfcc\0  := nop;

                ! set single precision floating mode, setf
                ! op code #170001
                setf\1  := nop;

                ! set single precision integer mode, seti
                ! op code #170002
                seti\2  := nop;

                ! unused op code #170003
                \3      nop

                END;

        \1      BEGIN
                DECODE fmsop =>

                ! unused op code #170010
                \0      nop;

                ! set double precision floating mode,
                ! setd op code #170011
                setd\1  := nop;

                ! set double precision integer mode, setl
                ! op code #170012
                setl\2  := nop;

                ! unused op code #170013
                \3      nop

                END
        );

fsingl\floating.point.single.operand.instructions :=
```

```
        (DECODE fd =>
        \0      BEGIN
                DECODE fuop =>

                ! clear floating, clrf op code #1704
                clrf\0  := ( (dcond => dclrf:=nop) NEXT dest);

                ! test floating, tstf op code #1705
                tstf\1  := ((dcond => dtstf:=nop) NEXT dest);

                ! form absolute value, absf op code #1706
                absf\2  := ( (dcond => dabsf:=nop) NEXT dest);

                ! negate floating, negf op code #1707
                negf\3  := ( (dcond => dnegf:= nop) NEXT dest)

                END;

        \1      BEGIN
                DECODE fuop =>

                ! clear floating, clrd op code #1704
                clrd\0  := ( (dcond => dclrd:= nop) NEXT dest);

                ! test floating, tstd op code #1705
                tstd\1  := ( (dcond => dtstd:= nop) NEXT dest);

                ! form absolute value, absd op code #1706
                absd\2  := ( (dcond => dabsd:= nop ) NEXT dest);

                ! negate floating, negd op code #1707
                negd\3  := ( (dcond => dnegd:= nop) NEXT dest)

                END
        );

fppcon\floating.point.processor.control :=
        (DECODE fuop =>

                ! floating status register setting instructions
                \0      fsrcon;

                ! load fpp processor status word, ldfps op code
                ! #1701
                ldfps\1 := ( (dcond => dldfps:= nop) NEXT dest);

                ! store fpp processor status word, stfps op code
                ! #1702
                stfps\2 := ( (dcond => dstfps:= nop) NEXT dest);

                ! store fpp status including exception address pointer,
                ! stst op code #1703
                stst\3  := ( (dcond => dstst:= nop) NEXT dest)
        );

fpext\floating.point.processor.isp:=
   (DECODE fd =>
        BEGIN
        DECODE fbop =>
        ! floating point processor mode control
        \0      fppcon;

        ! floating point unlary instructions
```

```
        \1      fsingl;

        ! floating multiply, mulf op code #1710x
        mulf\2  := ( (dcond => dmulf:= nop) NEXT dest);

        ! multiply and integerize floating, modf op code #1714x
        modf\3  := ( (dcond => dmodf:= nop) NEXT dest);

        ! floating add, addf op code #1720x
        addf\4  := ( (dcond => daddf:= nop) NEXT dest);

        ! load floating register, ldf op code #1724x
        ldf\5   := ( (dcond => didf:= nop) NEXT dest);

        ! floating subtract, subf op code #1730x
        subf\6  := ( (dcond => dsubf:= nop) NEXT dest);

        ! floating compare, cmpf op code #1734x
        cmpf\7  := ( (dcond => dcmpf:= nop) NEXT dest);

        ! store floating register, stf op code #1740x
        stf\10  := ( (dcond => dstf:= nop) NEXT dest);

        ! floating divide, divf op code #1744x
        divf\11 := ( (dcond => ddivf:= nop) NEXT dest);

        ! store floating exponent, stexp op code #1750x
        stexp\12:= ( (dcond => dstexp:= nop) NEXT dest);

        ! store and convert from floating to integer, stc op code #1754x
        \13     (DECODE fl =>
                        stcfl\0 := ( (dcond => dstcfl:= nop) NEXT dest);

                        stcfl\1 := ( (dcond => dstcfl:= nop) NEXT dest));

        ! convert from floating single to floating double precision,
        ! stcfd op code #1760x
        stcfd\14:= ( (dcond => dstcfd:= nop) NEXT dest);

        ! load floating exponent, ldexp op code #1764x
        ldexp\15:= ( (dcond => didexp:= nop) NEXT dest);

        ! load and convert from integer to floating, ldc op code #1770x
        \16     (DECODE fl =>
                        ldcif\0 := ( (dcond => didcif:= nop) NEXT dest);

                        ldcif\1 := ( (dcond => didcif:= nop) NEXT dest));

        ! load and convert from floating double to
        ! floating single, ldcdf op code #1774x
        ldcdf\17:= ( (dcond => didcdf:= nop) NEXT dest)

        END;

        BEGIN
        DECODE fbop =>
        ! floating point processor mode control
        \0      fppcon;

        ! floating point uniary instructions
        \1      fsingl;

        ! floating multiply, muld op code #1710x
```

```
        muld\2  := ( (dcond => dmuld:= nop) NEXT dest);

        ! multiply and integerize floating, modd op code #1714x
        modd\3  := ( (dcond => dmodd:= nop) NEXT dest);

        ! floating add, addd op code #1720x
        addd\4  := ( (dcond => daddd:=nop) NEXT dest);

        ! load floating register, ldd op code #1724x
        ldd\5   := ( (dcond => didd:=nop) NEXT dest);

        ! floating subtract, subd op code #1730x
        subd\6  := ( (dcond => dsubd:=nop) NEXT dest);

        ! floating compare, cmpd op code #1734x
        cmpd\7  := ( (dcond => dcmpd:=nop) NEXT dest);

        ! store floating register, std op code #1740x
        std\10  := ( (dcond => dstd:=nop) NEXT dest);

        ! floating divide, divd op code #1744x
        divd\11 := ( (dcond => ddivd:=nop) NEXT dest);

        ! store floating exponent, stexp op code #1750x
        stexpd\12:= ( (dcond => dstexd:=nop) NEXT dest);

        ! store and convert from floating to integer, stc op code #1754x
        \13     (DECODE fl =>
                        stcdi\0 := ( (dcond => dstcdi:=nop) NEXT dest);

                        stcdl\1 := ( (dcond => dstcdl:=nop) NEXT dest));

        ! convert from floating double to floating single precision,
        ! stcdf op code #1760x
        stcdf\14:= ( (dcond => dstcdf:=nop) NEXT dest);

        ! load floating exponent, ldexp op code #1764x
        ldexpd\15:= ( (dcond => didexd:=nop) NEXT dest);

        ! load and convert from integer to floating, ldc op code #1770x
        \16     (DECODE fl =>
                        ldcid\0 := ( (dcond => didcid:=nop) NEXT dest);

                        ldcld\1 := ( (dcond => didcld:=nop) NEXT dest));

        ! load and convert from floating single to
        ! floating double, ldcfc op code #1774x
        ldcfd\17:= ( (dcond => didcfd:=nop) NEXT dest)

        END
    );
```

```
classop\secondary.decode.into.classes:=
        BEGIN
        DECODE typeop =>

        ! subroutine/emulator trap
        \0      subemt;

        ! single operand class
        \1      singlop;

        ! shift operators
        \2      shiftop;

        ! unused op codes
        \3      nop
        END;

extop\extended.op.codes:=
        BEGIN
        DECODE jetop =>
        ! integer extented instructions
        \0      intext;

        ! floating point instructions
        \1      fpext
        END;


reserop\reserve.op.code:=
        BEGIN
        DECODE resop =>
        \0      branop;

        \1      classop
        END;
```

```
exec\instruction.execution:=
        BEGIN
        DECODE bop =>
        ! reserved op code
        \0      reserop;
        ! move and move byte
        ! mov op code #01, movb op code #11
        IMOV\1:=        BEGIN
                (DECODE byop =>
                        ( MOV:=nop NEXT
                        (scond => (smov:=nop)) NEXT
                        ( dcond => (dmov:=nop)));
                        ( MOVB:=nop NEXT
                        (scond => (smovb:=nop)) NEXT
                        ( dcond => (dmovb:=nop)))
                ) NEXT
                source NEXT
                temp ← src NEXT
cmov:=          (v ← 0) NEXT
                setncc NEXT
                setzcc NEXT
                dest NEXT
                mbr ← temp<15:0> NEXT
                write
                END;

        ! compare and compare byte
        ! cmp op code #02
        ! cmpb op code #12
        ICMP\2:=        BEGIN
                (DECODE byop =>
                        ( CMP:=nop NEXT
                        (scond => (scmp:=nop)) NEXT
                        ( dcond => (dcmp:=nop)));
                        ( CMPB:=nop NEXT
                        (scond => (scmpb:=nop)) NEXT
                        ( dcond => (dcmpb:=nop)))
                ) NEXT
                source NEXT
                dest NEXT
                read NEXT
                (DECODE byop =>
                \0      temp ← (src<15:0> + (NOT mbr) + 1)<16:0>;
                \1      temp ← (src<7:0> + (NOT mbr<7:0>) + 1)<8:0>
                ) NEXT
                setncc NEXT
                setzcc NEXT
ccmp:=          ((DECODE byop => c ← NOT temp<16>; c ← NOT temp<8>) NEXT
                (DECODE byop =>
                \0      v ← (temp<15> eqv mbr<15>) and (src<15> xor mbr<15>);
                \1      v ← (temp<7> eqv mbr<7>) and (src<7> xor mbr<7>)
                        ))
                END;

        ! bit test and bit test byte
        ! bit op code #03, bitb op code #13
        IBIT\3:=        BEGIN
                (DECODE byop =>
                        ( BIT:=nop NEXT
                        (scond => (sbit:=nop)) NEXT
                        ( dcond => (dbit:=nop)));
                        ( BITB:=nop NEXT
                        (scond => (sbitb:=nop)) NEXT
```

```
                        ( dcond => (dbitb:=nop)))
                ) NEXT
                source NEXT
                dest NEXT
                read NEXT
                temp ← src and mbr NEXT
                setncc NEXT
                setzcc NEXT
cbit:=          (v ← 0)
                END;

        ! bit clear and bit clear byte
        ! bic op code #04, bicb op code #14
        IBIC\4:=        BEGIN
                (DECODE byop =>
                        ( BIC:=nop NEXT
                        (scond => (sbic:=nop)) NEXT
                        ( dcond => (dbic:=nop)));
                        ( BICB:=nop NEXT
                        (scond => (sbicb:=nop)) NEXT
                        ( dcond => (dbicb:=nop)))
                ) NEXT
                source NEXT
                dest NEXT
                read NEXT
                temp ← (not src<15:0>) and mbr NEXT
                setncc NEXT
                setzcc NEXT
cbic:=          (v ← 0) NEXT
                mbr ← temp<15:0> NEXT
                write
                END;

        ! bit set and bit set byte
        ! bis op code #05, bisb op code #15
        IBIS\5:=        BEGIN
                (DECODE byop =>
                        ( BIS:=nop NEXT
                        (scond => (sbis:=nop)) NEXT
                        ( dcond => (dbis:=nop)));
                        ( BISB:=nop NEXT
                        (scond => (sbisb:=nop)) NEXT
                        ( dcond => (dbisb:=nop)))
                ) NEXT
                source NEXT
                dest NEXT
                read NEXT
                temp ← src<15:0> or mbr NEXT
                setncc NEXT
                setzcc NEXT
cbis:=          (v ← 0) NEXT
                mbr ← temp<15:0> NEXT
                write
                END;

        ! add and subtract
        \6      BEGIN
                DECODE byop =>
                ! add, add op code #06
                ADD\0:= BEGIN
                        (scond => (sadd:=nop)) NEXT
                        (dcond => (dadd:=nop)) NEXT
                        source NEXT
```

```
                        dest NEXT
                        read NEXT
                        temp ← (src<15:0> + mbr)<16:0> NEXT
cadd:=                  (v ← (src<15> eqv mbr<15>) and (src<15> xor temp<15>) NEXT
                        c ← temp<16>) NEXT
                        setncc NEXT
                        setzcc NEXT
                        mbr ← temp<15:0> NEXT
                        write
                        END;

                ! subtract, sub op code #16
                SUB\1:= BEGIN
                        byop ← 0 NEXT
                        (scond => (ssub:=nop)) NEXT
                        (dcond => (dsub:=nop)) NEXT
                        source NEXT
                        dest NEXT
                        read NEXT
                        temp ← (mbr + (NOT  src)<15:0> +1)<16:0> NEXT
csub:=                  (v ← (src<15> xor mbr<15>) and (src<15> eqv temp<15>) NEXT
                        c ← NOT temp<16>) NEXT
                        setncc NEXT
                        setzcc NEXT
                        mbr ← temp<15:0> NEXT
                        write
                        END
                END;

        ! extended instruction set
        \7      extop
        END


ERALCED
```

PDP-11 ISP DESCRIPTION 16-1

```
!main sequence of the isp description
!-----------------------------------

!instruction interpretation process
!-----------------------------------

inter\instruction.interpretation:=
        BEGIN
        IF run =>
                mar ← pc NEXT
                (IF sr0<15:13> eql 0 => sr2 ← mar<15:0>) NEXT
                byop ← 0 NEXT
                read NEXT
                ir ← mbr; pc ← (pc + 2)<15:0> NEXT
                byop ← i<15> NEXT
                exec NEXT
                inter
        END


)       !end of description
```

TABLE OF CONTENTS

1. Sample Simulation Run

The following is a transcript of a typical session using the ISP simulator. The session consists of running one of the benchmarks (Bit Test, Set, and Reset) on the PDP-11. A listing of the benchmark program appears after the session. Comments have been added for clarity.

The input for a simulation session consists of several files prepared off-line. These files include: The benchmark program (derived from the assembly listing), a driver (simulation commands used to initialize the parameters for the benchmark), A command file with a list of unimplemented instructions (these must be trapped), and finally, a command file with a list of those ISP procedures which must be "opaqued" (these are the procedures during which the activity counters are disabled).

```
ru pdp11m
ISP SIMULATOR V3 - NRL ARF STAGE 2
Friday 10 Sep 76 17:13:50 PDP11M.ISP[L410MB25]
SERIALIZATION COMPLETED
SPACE ALLOCATED
TYPE HELP FOR HELP
TYPE <ESC> TO INTERRUPT SIMULATION LOOPS

>read fadl.sim                  ! Read in the benchmark file

>>RADIX OCTAL

>>DECHO                         ! The benchmark file disables the listing
                                ! on the user terminal.

>>100 LINES READ

>read fa.dr3                    ! Read in the driver file

>>!     HERE COMES THE DRIVER (CALLS)

>>

>>SETVAL MW[3000]=013746 005202        !     MOV     @#5202,-(SP)    ; F

>>SETVAL MW[3002]=013746 005204        !     MOV     @#5204,-(SP)    ; N

>>SETVAL MW[3004]=012746 004000        !     MOV     #4000,-(SP)     ; A1
```

```
>>SETVAL MW[3000]←012746 005200          !       MOV     #5200,-(SP)     ; RC

>>SETVAL MW[3010]←012746 005206          !       MOV     #5200,-(SP)     ; W

>>SETVAL MW[3012]←004737 001000          !       JSR     PC,@#1000       ; BTSR

>>SETVAL MW[3014]←000000                 !       HLT

>>      ! The above sequence of PDP-11 instructions push the parameters
        ! onto the stack, call the benchmark as a routine, and halt.


>>SETVAL MW[2000]←123457 071234 167000 145670   !       BIT STRING

>>SETVAL MW[2500]←0                      !       RETURN CODE

>>SETVAL MW[2501]←2                      !       F

>>SETVAL MW[2502]←25                     !       N

>>SETVAL MW[2503]←0                      !       WORK AREA

>>

>>SETVAL PC←6000

>>SETVAL SP←200

        ! The above sequence initializes the data (parameters), the stack
        ! pointer and the program counter (which now points to the code
        ! sequence that pushes the parameters and call the routine.
>>SETVAL A←0     ! This is a ISP internal variable - indicates whether the
                 ! machine is running, halted, or waiting.
>>! RUNNING

>>SETCTR ALL 0,0
>>      ! RESET COUNTERS

>>READ OPQ11.SIM[L410MB25]
>>>! PDP11 OPAQUED PROCEDURES

>>>DECHO

>>>53 LINES READ

>>READ UUO11.SIM[L410MB25]
>>>! UNIMPLEMENTED OPERATION BREAKS

>>>DECHO

>>>15 LINES READ

>>TRACE IR,PC,R,MWIO              ! Trace a few selected registers
                                  ! IR is the Instruction Register,
                                  ! PC is the Program Counter (R[7]),
```

```
                                    ! R[0:7] are the general registers,
                                    ! MWIO is the I/O page (R is mapped onto MWIO)


>>BREAK JSR,RTS                     ! Break on selected instructions

>>26 LINES READ

>start inter                        ! Here we start the simulation

@ INTER   +#15    IR      =#13746
@ INTER   +#20    PC      =#6002
@ SINCD   +#22    R       [#7]=#6004
@ DDECRD  +#21    R       [#6]=#176
@ INTER   +#15    IR      =#13746
@ INTER   +#20    PC      =#6006
@ SINCD   +#22    R       [#7]=#6010
@ DDECRD  +#21    R       [#6]=#174
@ INTER   +#15    IR      =#12746
@ INTER   +#20    PC      =#6012
@ SINCD   +#22    R       [#7]=#6014
@ DDECRD  +#21    R       [#6]=#172
@ INTER   +#15    IR      =#12746
@ INTER   +#20    PC      =#6016
@ SINCD   +#22    R       [#7]=#6020
@ DDECRD  +#21    R       [#6]=#170
@ INTER   +#15    IR      =#12746
@ INTER   +#20    PC      =#6022
@ SINCD   +#22    R       [#7]=#6024
@ DDECRD  +#21    R       [#6]=#166
@ INTER   +#15    IR      =#4737
@ INTER   +#20    PC      =#6026
BREAK AFTER JSR                     ! The simulation stops on a breakpoint

*setctr all 0,0                    ! The real benchmark starts here, we must
                                    ! reset all counters (they were modified
                                    ! during the benchmark calling sequence)

*cont                               ! We continue the simulation

@ DINCRD  +#22    R       [#7]=#6030
@ JSR     +#14    R       [#7]=#6030
@ JSR     +#15    PC      =#1000
@ INTER   +#15    IR      =#10046
@ INTER   +#20    PC      =#1002
@ DDECRD  +#21    R       [#6]=#162
@ INTER   +#15    IR      =#10146
@ INTER   +#20    PC      =#1004
@ DDECRD  +#21    R       [#6]=#160
@ INTER   +#15    IR      =#5076
@ INTER   +#20    PC      =#1006
@ DINDD   +#6     PC      =#1010
@ INTER   +#15    IR      =#16660
@ INTER   +#20    PC      =#1012
@ SINDD   +#6     PC      =#1014
@ WRITE   +#131   MWIO    [#3740000]=#25
@ INTER   +#15    IR      =#42700
```

```
@ INTER  +#20   PC     =#1016
@ SINCD  +#22   R      [#7]=#1020
@ WRITE  +#131  MWIO   [#3740001]=#5
@ INTER  +#15   IR     =#12701
@ INTER  +#20   PC     =#1022
@ SINCD  +#22   R      [#7]=#1024
@ WRITE  +#131  MWIO   [#3740001]=#1
@ INTER  +#15   IR     =#72100
@ INTER  +#20   PC     =#1026
@ CVASH  +#7    R      [#1]=#40
@ INTER  +#15   IR     =#16600
@ INTER  +#20   PC     =#1030
@ SINDD  +#6    PC     =#1032
@ WRITE  +#131  MWIO   [#3740001]=#25
@ INTER  +#15   IR     =#72027
@ INTER  +#20   PC     =#1034
@ DINCRD +#22   R      [#7]=#1036
@ CVASH  +#7    R      [#0]=#2
@ INTER  +#15   IR     =#66600
@ INTER  +#20   PC     =#1040
@ SINDD  +#6    PC     =#1042
@ WRITE  +#131  MWIO   [#3740001]=#4002
@ INTER  +#15   IR     =#130110
@ INTER  +#20   PC     =#1044
@ INTER  +#15   IR     =#1402
@ INTER  +#20   PC     =#1046
@ BRANCH +#5    PC     =#1052
@ INTER  +#15   IR     =#22766
@ INTER  +#20   PC     =#1054
@ SINCD  +#22   R      [#7]=#1056
@ DINDD  +#6    PC     =#1060
@ INTER  +#15   IR     =#1405
@ INTER  +#20   PC     =#1062
@ BRANCH +#5    PC     =#1074
@ INTER  +#15   IR     =#150110
@ INTER  +#20   PC     =#1076
@ INTER  +#15   IR     =#773
@ INTER  +#20   PC     =#1100
@ BRANCH +#5    PC     =#1066
@ INTER  +#15   IR     =#12601
@ INTER  +#20   PC     =#1070
@ SINCD  +#22   R      [#6]=#162
@ WRITE  +#131  MWIO   [#3740001]=#0
@ INTER  +#15   IR     =#12600
@ INTER  +#20   PC     =#1072
@ SINCD  +#22   R      [#6]=#164
@ WRITE  +#131  MWIO   [#3740001]=#0
@ INTER  +#15   IR     =#207
@ INTER  +#20   PC     =#1074
BREAK AFTER RTS                  ! the simulation stops at the end of the
                                 ! benchmark (the return instruction)

#outctr fadl.rm3                 ! we dump all the counters into a file

#cont                            ! we continue the simulation

@ RTS    +#2    PC     =#1074
@ RTS    +#7    R      [#7]=#6030
```

```
e INTER  +#15   IR      =#0
e INTER  +#20   PC      =#6032
SIMULATION COMPLETED              ! we executed the Halt instruction

RUN TIME(10 usec units)=831678
RTM OPS EXECUTED=4535

>exit                             ! we finish the session

EXIT
```

```
RADIX OCTAL
DECMD
!CFAF   MACM11   V003F   5-JUL-76   12:54   PAGE 1
!BTSR1  M11
!
!      1                                    00100          .TITLE  CFAF
!      2                                    00200  ; Bit test, set, or reset subroutine
!      3                                    00300  ; CFA program F,  CMU programmer 3 -
!      4                                    00400  ; 8 June 1976
!      5                                    00500          .GLOBL  BTSR
!      6                 000000             00600  R0=%0
!      7                 000006             00700  SP=%6
!      8                 000007             00800  PC=%7
!      9                                    00900  ;
!     10                                    01000  ; I assume that bits are numbered fr
!     11                                    01100  ; of a word.
!     12                                    01200  ;
!     13                                    01300  ; Offsets of parameters from stack p
!     14                                    01400  ;
!     15                 000004             01500  SAVE=4           ; we need to save 2
!     16                                    01600  ;
!     17                 000016             01700  F=12.+SAVE       ; function code
!     18                 000014             01800  N=10.+SAVE       ; relative bit numbe
!     19                 000012             01900  A1=6.+SAVE       ; address of bit str
!     20                 000010             02000  RC=4.+SAVE       ; address of return
!     21                 000006             02100  WORK=2.+SAVE     ; address of work ar
!     22                                    02200  ;
!     23         000000'                    02300  BTSR:
!     24         000000' 010046             02400          MOV     R0,-(SP)
!     25         000002' 010146             02500          MOV     R1,-(SP)
!     26         000004' 005076  000010     02600          CLR     @RC(SP)         ; ze
!     27         000010' 016600  000014     02700          MOV     N(SP),R0        ; ge
!     28         000014' 042700  177770     02800          BIC     #177770,R0      ; th
!     29         000020' 012701  000001     02900          MOV     #1,R1
!     30         000024' 072100             03000          ASH     R0,R1           ; sh
!     31         000026' 016600  000014     03100          MOV     N(SP),R0
!     32         000032' 072027  177775     03200          ASH     #-3,R0          ; by
!     33         000036' 066600  000012     03300          ADD     A1(SP),R0       ; th
!     34         000042' 130110             03400          BITB    R1,@R0
!     35         000044' 001402             03500          BEQ     L1
!     36         000046' 005276  000010     03600          INC     @RC(SP)         ; th
!     37         000052' 022766  000002  000016  03700  L1:     CMP     #2,F(SP)        ; se
!     38         000060' 001405             03800          BEQ     SET
!     39         000062' 100001             03900          BPL     QUIT
!     40         000064' 140110             04000          BICB    R1,@R0          ; FC
!     41         000066' 012601             04100  QUIT:    MOV     (SP)+,R1        ; ex
!     42         000070' 012600             04200          MOV     (SP)+,R0
!     43         000072' 000207             04300          RTS     PC
!     44         000074' 150110             04400  SET:    BISB    R1,@R0          ; FC
!     45         000076' 000773             04500          BR      QUIT
!     46                 000001             04600          .END
!CFAF   MACM11   V003F   5-JUL-76   12:54   PAGE 1-1
!BTSR1  M11
!
```

```
!A1     = 000012        BTSR    000000RG        F       = 000016        L1      000052R
!N      = 000014        PC      =X000007        QUIT    000066R         RC      = 000010
!R0     =X000000        R1      =X000001        R2      =X000002        R3      =X000003
!R4     =X000004        R5      =X000005        R6      =X000006        R7      =X000007
!SAVE   = 000004        SET     000074R         SP      =X000000        WORK    = 000006
!.MACN. = 000003        .       = 000100R
!
!
!CFAF   MACN11   V003F   5-JUL-76   12:54   PAGE 1-2
!BTSR1  M11
!
!
!
! ERRORS DETECTED: 0
!
! *btsr1,btsr1=btsr1
! TOTAL # OF PST ACCESSES = 22
! TOTAL # OF 11/45 INSTRUCTIONS = 2
! RUN-TIME:  1 SECONDS
! CORE USED:  4K
!
                                ! Here begin the simulation commands
                                ! derived from the above listing
                                ! relocation address = word 400 (octal) = byte 1000
!
SETVAL MW[400]=010046
SETVAL MW[401]=010146
SETVAL MW[402]=005076 000010
SETVAL MW[404]=016600 000014
SETVAL MW[406]=042700 177770
SETVAL MW[410]=012701 000001
SETVAL MW[412]=072100
SETVAL MW[413]=016600 000014
SETVAL MW[415]=072027 177775
SETVAL MW[417]=066600 000012
SETVAL MW[421]=130110
SETVAL MW[422]=001402
SETVAL MW[423]=005276 000010
SETVAL MW[425]=022766 000002 000016
SETVAL MW[430]=001405
SETVAL MW[431]=100001
SETVAL MW[432]=140110
SETVAL MW[433]=012601
SETVAL MW[434]=012600
SETVAL MW[435]=000207
SETVAL MW[436]=150110
SETVAL MW[437]=000773

ECHO
```

IV.III.3. Simulation Command Files: Driver Program

```
!       HERE COMES THE DRIVER (CALLS)

SETVAL MW[3000]←013746 005202   !       MOV     @#5202,-(SP)    ; F
SETVAL MW[3002]←013746 005204   !       MOV     @#5204,-(SP)    ; N
SETVAL MW[3004]←012746 004000   !       MOV     #4000,-(SP)     ; A1
SETVAL MW[3006]←012746 005200   !       MOV     #5200,-(SP)     ; RC
SETVAL MW[3010]←012746 005206   !       MOV     #5206,-(SP)     ; W
SETVAL MW[3012]←004737 001000   !       JSR     PC,@#1000       ; BTSR
SETVAL MW[3014]←000000          !       HLT

SETVAL MW[2000]←123457 071234 167000 145670     !       BIT STRING
SETVAL MW[2500]←0               !       RETURN CODE
SETVAL MW[2501]←2               !       F
SETVAL MW[2502]←25              !       N
SETVAL MW[2503]←0               !       WORK AREA

SETVAL PC←6000
SETVAL SP←200
SETVAL A←0      ! RUNNING
SETCTR ALL 0,0          ! RESET COUNTERS
READ DPQ11.SIM[L410MB25]
READ UUQ11.SIM[L410MB25]
TRACE IR,PC,R,MWIO
BREAK JSR,RTS
```

IV.III.5. Simulation Command Files: Opaqued Procedures

```
! PDP11 OPAQUED PROCEDURES
DECHO
OPAQUE CADC
OPAQUE CADD
OPAQUE CASH
OPAQUE CASHB
OPAQUE CASHBC
OPAQUE CASHC1
OPAQUE CASHC2
OPAQUE CASHCB
OPAQUE CASL
OPAQUE CASLB
OPAQUE CASR
OPAQUE CBIC
OPAQUE CBIS
OPAQUE CBIT
OPAQUE CCCO
OPAQUE CCLR
OPAQUE CCMP
OPAQUE CCOM
OPAQUE CEXOR
OPAQUE CMOV
OPAQUE CNASHC
OPAQUE CNEG
OPAQUE CROL
OPAQUE CROLB
OPAQUE CROR
OPAQUE CRORB
OPAQUE CSBC
OPAQUE CSUB
OPAQUE CSWAP
OPAQUE CSXT
OPAQUE CTST
OPAQUE CVASH
OPAQUE CVASHC
OPAQUE CVASL
OPAQUE CVASR
OPAQUE CVROL
OPAQUE CVROR
OPAQUE CZASHC
OPAQUE SETNCC
OPAQUE SETVCC
OPAQUE SETZCC
OPAQUE SIGNEX
OPAQUE CNMUL
OPAQUE CZMUL
OPAQUE CVMUL
OPAQUE CCMUL
OPAQUE CNDIV
OPAQUE CZDIV
OPAQUE CVDIV
OPAQUE CCDIV
ECHO
```

IV.III.4. Simulation Command Files: Unimplemented Instructions

```
! UNIMPLEMENTED OPERATION BREAKS
DECHO
BREAK MFP       ! MFPI,MFPD
BREAK MTP       ! MTPI,MTPD
BREAK SPL
BREAK MUL
BREAK DIV
BREAK FPEXT     ! CFCC,SETF,SETI,SETD,SETL
                ! CLRF,TSTF,ABSF,NEGF,CLRD,TSTD,ABSD,NEGD
                ! DFPS,STFPS,STST
                ! MULF,MODF,ADDF,LDF,SUBF,CMPF,STF,DIVF,STEXP,
                ! STCFI,STCFL,STCFD,LDEXP,LDCIF,LDCLF,LDCDF,
                ! MULD,MODD,ADDD,LDD,SUBD,CMPD,STD,DIVD,STEXP,
                ! STCDI,STCDL,STCDF,LDEXPD,LDCID,LDCLD,LDCDF
ECHO
```